AN1442: SiWx917 SoC Secure Boot with Anti-Rollback Protection

This application note describes the design of Secure boot with anti-rollback protection on SiWx917 devices. It also provides examples of how to implement the Secure Boot process and protect against firmware rollback.

For more information on provisioning the SiWx917 for secure boot, see the following: UG574: SiWx917 SoC Manufacturing Utility User Guide.

KEY FEATURES

- Describes the secure boot process for SiWx917 devices
- Describes the anti-rollback mechanism
- Describes the layout of RPS files
- Provides examples of configuring SiWx917 devices for secure boot

Table of Contents

1. SiWx917 Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options on the SiWx917 to help developers build secure devices, secure application software, and secure paths of communication to manage those devices.

The SiWx917 consists of the following 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.
- Encrypted XiP: Process that adds confidentiality when instructions are being executed in place from off-die or off-chip storage.

It is also possible to lock access to the debug ports for operational security, and to unlock them when access is required.

User Assistance

The following table summarizes the key security documents:

Table 1.1. Documents Related to Secure Boot for this Product

Document	Summary
AN1431: SiWx917 SoC Firmware Update Application Note	Describes how to perform SoC firmware updates
AN1416: SiWG917 SoC Memory Map Application Note	Describes the SiWG917 SoC Memory Map
AN1439: SiWx917 Hardware Debugging Guidelines	Guidelines for debugging hardware related issues with SiWx917
AN1428: SiWx917 Debug Lock	How to lock and unlock SiWx917 debug access ports
AN1442: SiWx917 SoC Secure Boot with Anti-Rollback Protection	Describes the secure boot and anti-rollback protection processes on SiWx917
UG162: Simplicity Commander Reference Guide	Describes commands available in Simplicity Commander for provisioning eFuses for secure boot and anti-rollback protection
UG574: SiWx917 SoC Manufacturing Utility User Guide	Describes steps for provisioning SiWx917 hardware for production

Key Reference

Secure boot requires the use of cryptographic keys for authenticating firmware before allowing it to run. The following table summarizes the cryptographic keys used for secure boot and their intended purpose:

Table 1.2. Secure Boot Keys

Key identifier	Description	Key Type	Keysize (bits)	Storage	Lifetime
Master key	Used for decrypting and authenticating keys used by the NWP core	Symmetric, AES	256	Intrinsic[3]	Permanent
Unwrap key	Used for decrypting and authenticating keys used by the M4 core	Symmetric, AES	256	Intrinsic[3]	Permanent
TA public key[1,2]	Validates TA firmware	Asymmetric, ECC	256	Flash	Permanent
TA OTA key[2]	Encrypt/decrypt TA firmware, generate CMAC MIC	Symmetric, AES	256	Flash	Permanent
M4 public key[1,2]	Validates M4 firmware	Asymmetric, ECC	256	Flash	Updatable
M4 OTA key[2]	Encrypt/decrypt M4 firmware, generate CMAC MIC	Symmetric, AES	256	Flash	Updatable

Note:

1. Private keys must be kept secure and should be stored as securely as possible.
2. These keys are wrapped for tamper resistance
3. Intrinsic keys are generated at runtime using the PUF and a 52 byte key code, stored in flash.

eFuse Reference

Secure boot features in the SiWx917 are set in one-time programmable flags, known as eFuses. These efuses are one-time programmable and should be only written to after development is complete. During development, these options should be set using the Master Boot Record (MBR).
Master Boot Record (MBR) is stored in flash and contains information like clock frequencies, offsets of structures like eFuse copy, SPI configurations, External Flash details, etc. There are separate MBRs for TA and M4 at the beginning of their respective flash regions. Any SiWx917 IC that is shipped out of the factory will have a default MBR. Using the OPN of a particular device, the user can update the MBR. For more information on manipulating the MBR, consult UG574: SiWx917 SoC Manufacturing Utility User Guide.

The e-fuse settings relevant to secure boot are summarized in the following table:

Table 1.3. Secure Boot eFuses

eFuse Name	Description
m4_secure_boot_enable	Enables MIC-based validation of firmware images before executing
m4_anti_roll_back	Enables anti-rollback protection for the M4 firmware
m4_digital_signature_validation	Enables authentication of M4 firmware before executing
ta_secure_boot_enable	Enables MIC-based validation of the TA firmware before executing
ta_anti_roll_back	Enables anti-rollback protection for the TA firmware
ta_digital_signature_validation	Enables authentication of TA firmware before executing
disable_m4_jtag	Locks the JTAG port of the M4 core
disable_ta_jtag	Locks the JTAG port of the TA core
safe_upgrade_frm_host	Enables failsafe upgrade mode. Firmware updates do not overwrite existing firmware until they've been authenticated.

2. Secure Boot Process

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 often can be subverted through the injection of malicious code.

Secure Boot works to ensure 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 halt in the bootloader until an authentically signed image can be loaded onto it. While 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. In the SiWx917, the root of trust (i.e., security bootloader) is in ROM.

SiWx917 devices use a security bootloader, which runs from ROM on the Network Processor (NWP) core and an application bootloader which runs on the application core (M4). The security bootloader is responsible for verifying the integrity and authenticity of the NWP and M4 firmware in flash before allowing it to be loaded by the application bootloader.

Initializing PUF and Generating Intrinsic Keys in SiWx917

The SiWx917 uses a physically unclonable function (PUF) circuit for creating a unique-per-device encryption key. The PUF on each device must be initialized to function. Once initialized, a PUF activation code is stored in the NWP flash. The activation code is 1192 bytes in size. During initialization, several so-called intrinsic keys are derived from the PUF. Each of these intrinsic keys is 256 bits in length and is stored in the form of a 52 byte key code in the NWP flash. The intrinsic keys relevant to secure boot are summarized in the table below.

Table 2.1. Intrinsic Keys Used for Secure Boot

Key identifier	Use
Master key	Used to wrap and unwrap keys used by the NWP core
Unwrap key	Used to wrap and unwrap keys used by the M4 core

ECDSA-P256 Secure Boot in SiWx917

Firmware running on the SiWx917 can be validated for authenticity with a digital signature (ECDSA-P256), validated for integrity using a message integrity check (MIC), or both. The options can be set either in the MBR or in the device's eFuses depending on the user's development journey. Important points to note -

- If the 'secure_boot_enable' eFuse is set for a particular core, the firmware image for the core must include a valid MIC.
- If the 'digital_signature_validation' eFuse is set for a particular core, the firmware image must include a valid signature.

Silicon Labs recommends the use of ECDSA-P256 signatures for validating firmware images since this method is more secure than a MIC. ECDSA is a stronger method of protection since it uses an asymmetric key and the private signing key does not exist on the device. Using MIC only for validating firmware is a weaker method of protection since it uses a symmetric key that does exist on the device. If the key is leaked, unauthorized users will be able to calculate a valid MIC for their own firmware. However, if performance is critical, the use of a MIC for validating firmware offers some protection.

Once a firmware image has been built, an RPS file is created which can be flashed to the device. As part of the creation of the RPS file, a MIC can be calculated with the AES-CMAC algorithm using the OTA key. The OTA key for NWP is used to calculate the MIC for the NWP firmware and the M4 OTA key is used to calculate the MIC for the M4 firmware. In addition, a message digest can be produced from the firmware image using one of the supported hash algorithms, SHA2/256, SHA2/384 or SHA2/512. The message digest for M4 firmware is then signed to produce an ECDSA-P256 signature. The M4 private key is used to sign images for the M4 core and the NWP private key is used to sign images for the TA core. Signatures for either image are verified by the security bootloader using the corresponding M4 or NWP public key which is stored wrapped in flash on the device. The M4 core is held in reset until a valid image is loaded. Once the image is successfully verified, it can be loaded and allowed to run. If the image verification fails, the M4 core will be held in reset, halting execution.

Note: If the SiWx917 is operating with an unsigned firmware image at the time when secure boot is enabled, the device will continue to run that firmware image post-enablement of secure boot. All subsequent firmware upgrade images must be signed. Silicon Labs' recommends erasing any prior unsigned images on the device after enabling secure boot during manufacturing.

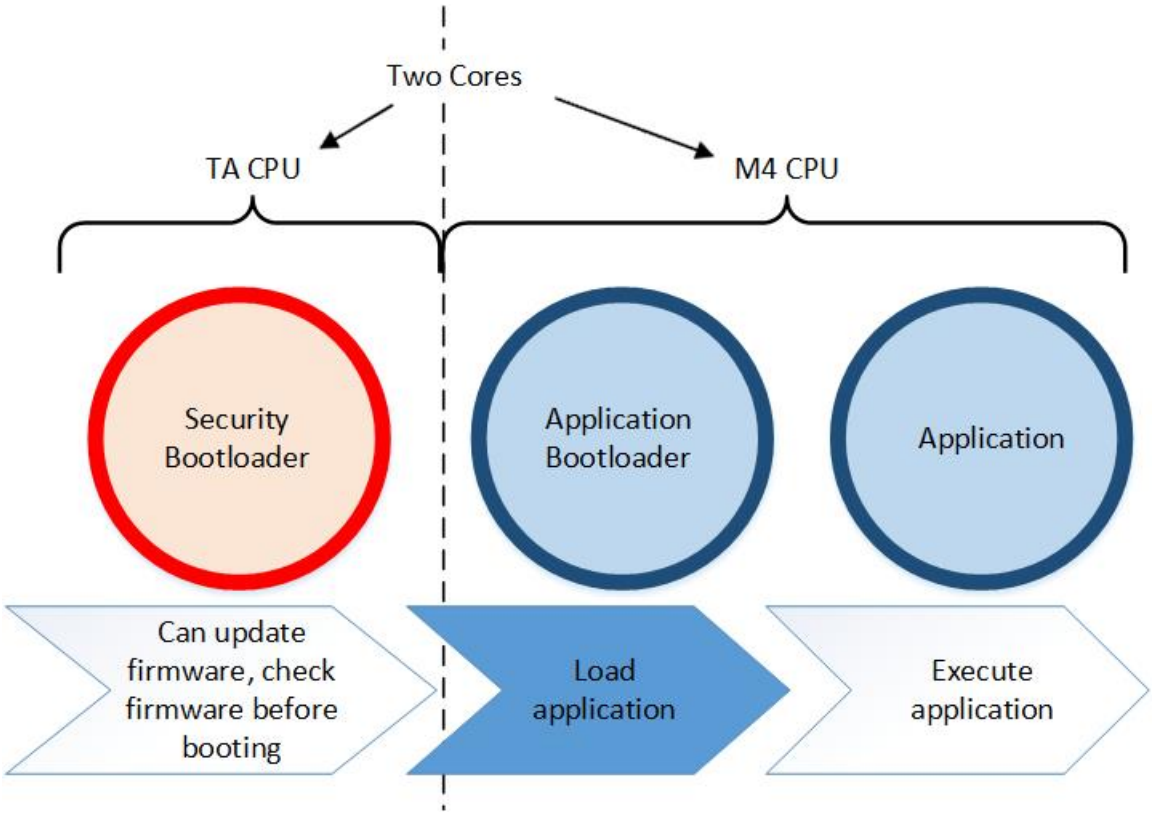


Figure 2.1. Secure Boot Process in SoC Mode

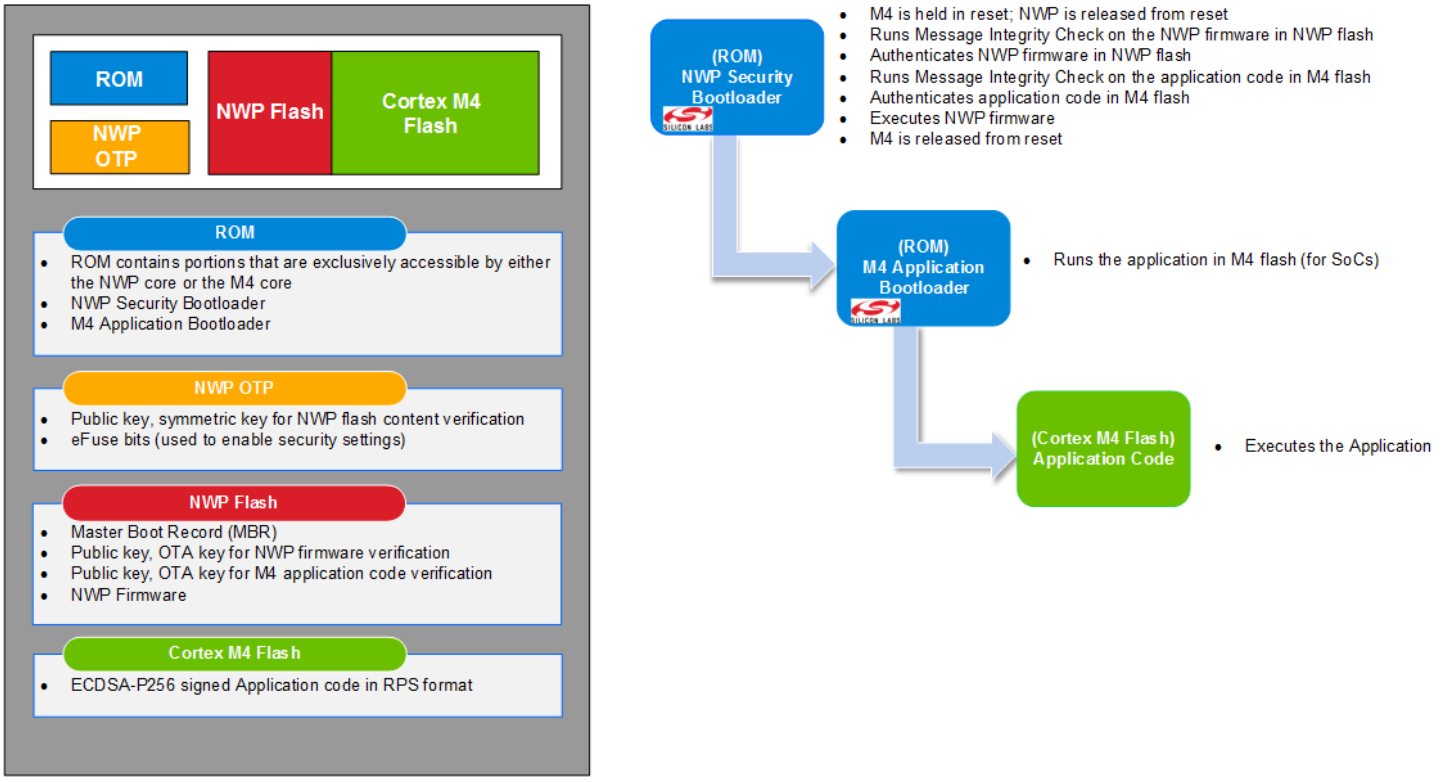


Figure 2.2. Secure Boot Flow

Security Bootloader

The security bootloader executes from ROM, is accessed by the NWP core, and is maintained by Silicon Labs.

Upon reset, execution always begins with the security bootloader which performs the following functions:

1. Configures hardware based on the contents of the MBR and eFuse settings
2. Optionally, performs In-system programming (ISP) and firmware upgrades
3. Authenticates firmware in flash
4. Starts the application bootloader

Application Bootloader

Once started by the security bootloader, the application bootloader, also executing from ROM, is accessed by the M4 core to load the M4 application and start executing it.

Flash Protection Levels

The security bootloader provides 3 flash protection levels which can be used to secure different sections of the Flash for different purposes:

Table 2.2. Flash Protection Levels

Protection level	MBR field	Unlock condition	Address range
Level 1	flash_level1_protect_bits	Never	4 KB
Level 2	flash_level2_protect_bits	After firmware update	64 KB
Level 3	flash_level3_protect_bits	During firmware update	2 MB
Note: Level 1 region is provisioned at manufacturing time and cannot be changed after that.			

Anti-Rollback Protection

Anti-rollback protection is designed to prevent the SiWx917 from downgrade attacks. Downgrade attacks exploit vulnerabilities that are patched in more recent versions of firmware by downgrading that firmware to an older version that contains known vulnerabilities. It is important to protect against downgrade attacks to ensure that if an issue is patched, it cannot be exploited again once the firmware is upgraded to a patched version. Anti-rollback protection can be enabled for the NWP or M4 core by setting the appropriate eFuse bit. See the eFuse Reference Table, Table 1.3 Secure Boot eFuses on page 5 for more information. For a detailed description of how firmware updates are applied, please refer to section 3 of AN1431.

Firmware versions are composed as follows:

Table 2.3. Firmware Version Layout in RPS Files

Major Version	Minor Version	Security Version	Build Number
8 bits	8 bits	8 bits	8 bits

The version of any application upgrade is checked for the following before applying the upgrade:

- All 32 bits are checked to verify that the new image is greater than the currently installed image
- The security version is checked to verify that it is greater than the security version of the currently installed image

The application version, which contains Major, Minor and Build number versions (in the format Major:Minor:Build), is stored in the Flash Memory Controller (FMC). Firmware updates can take place when only the application version increases, the security version increases or both.

In case the anti-rollback check fails, the firmware update is not copied to the application space to replace the existing firmware and the existing firmware image continues to execute.

Once an image has been successfully loaded, OTP storage will be updated with the security version if the 'anti_roll_back' eFuse has been set for the core that the image runs on. The security version of the current firmware for each core is stored separately in OTP. The security version is represented by setting the number of bits in OTP. For the NWP core, the security version can be between 1 and 128, for the M4 core, the security version can be between 1 and 64.

Table 2.4. Security Version Storage in OTP and RPS Files

OTP											RPS File
MSB	…	8	7	6	5	4	3	2	1	0	Security version
0	0	0	0	0	0	0	0	0	0	1	0x01
0	0	0	0	0	0	0	0	0	1	1	0x02
0	0	1	1	1	1	1	1	1	1	1	0x09

Example showing storage of security versions

Note: The MSB on the NWP core is bit 127, and the MSB on the M4 core is 63.

Authenticated Firmware Update Process

The security bootloader can validate the ECDSA-P256 signature of update images using the appropriate public key and/or the MIC using the appropriate AES key. The keys used are summarized in Table 1.2 Secure Boot Keys on page 4 in the ● Key Reference on page 4. Update images can be delivered in RPS format. Refer to AN1431: SiWx917 SoC Firmware Update for additional information. In case a firmware upgrade is determined to be invalid, either because of an invalid signature or MIC or because the firmware version is too low, the upgrade will not be applied and the previous firmware image remains in place.

3. RPS Format

The RPS file format is used for delivering update images which are optionally signed and encrypted. An RPS file may contain a TA application image, M4 application image, or both. The format is specified in the table below:

Table 3.1. RPS File Layout

46 bytes	64 bytes	…	71 or 72 bytes
RPS header	Boot descriptors	Application binary (size varies)	Digital signature (optional) [1]

Note:
1. Signature is stored in ASN.1 format, not plain binary. The size of ASN.1 can vary depending on the content.

The RPS header is defined in the follow table.

Table 3.2. RPS Header Layout

Field	Size (bits)	Description
Control Flags	16	BIT(0) : 0 - NWPSS image 1 – M4 image BIT(1): 0 – No encryption is present 1 - Image is encrypted BIT(2): 0 - CRC based integrity check 1 - MIC based integrity check BIT(3): 0 – No digital signature 1 - Digitally signed (digital signature is located at the end of the RPS file) BIT(4): 0 – image will not be combined with another image 1 – image will be combined with another image BITS: (5-15) - Reserved
SHA size	16	Represents the SHA size used to compute the digest for the digital signature 0 - Reserved 1 - SHA_256 2 - SHA_384 3 - SHA_512
Magic word	32	0x900D900D
Image Size	32	Size of the binary image
Firmware Version	32	Firmware version number, See table 2.1 for the version layout
Flash Address	32	Address in flash to store the image
CRC	32	CRC of the image[1]
MIC	128	MIC of the image
Firmware Extended Version	32	Patch Version, Customer ID, Chip id, ROM Version

Field	Size (bits)	Description
Reserved	96	-
Magic bytes	32	0x900D900D

Note:
1. polynomial to use can be decided at the time of manufacturing

Table 3.3. Boot Descriptors Layout

Field	Size (bytes)	Description
Magic pattern	2	Pattern for identification of a valid Flash content (0x5aa5)
Offset	2	Offset of the binary image where the transfer should start from flash to RAM
IVT offset	4	Value to program VTOR register
Bootloader descriptor entries	56	Bootloader descriptor entries which are executed by the Bootloader while loading firmware - see Table 3.4 Bootloader Descriptors Layout on page 11 below for more details. Space is provided for 7 bootloader descriptor entries.

Table 3.4. Bootloader Descriptors Layout

Field	Size (bits)	Description
Length	24	Length of transfer to destination
Reserved	7	-
Last entry	1	If set, indicate it is last boot descriptor entry
Destination address	32	Destination address

Encrypted XIP

Encrypted execute-in-place (XIP) allows encrypted firmware images to be decrypted immediately before runtime. Encrypted XIP is useful when program and data are stored in external flash.

In-System Programming (ISP)

The secure firmware upgrade feature of the security bootloader checks the authenticity and integrity of the new firmware image, provided that these features have been enabled in the eFuse settings. The security bootloader will only update the image after successfully validating the authenticity of the image using the digital signature check and the integrity of the image using the MIC check, depending on which checks are enabled. If enabled, the anti-rollback feature will prevent downgrades to a lower firmware version, refer to section 2.x for more details. The security bootloader also supports OTA updates.

4. Secure Boot and Debug Lock

At the end of production, a critically important step is to lock the JTAG ports of both cores. Silicon Labs recommends that this step is completed at the end of production to prevent access or modification to the public keys and OTA keys that may be stored on the device. If any of these keys are modified after initial provisioning, secure boot will fail. For instructions on locking the debug port, refer to AN1428 : SIWX917 Debug Lock.

5. Examples

Overview

5.1 Provisioning for secure boot using Simplicity Commander

This section demonstrates how to provision the security settings onto a factory new SiWx917 device. The most reliable method of provisioning settings is to place the device into ISP mode. For instance, hold the ISP button on the radioboard (BRD4338A), and press and release the reset button on the WSTK/WPK board before releasing the ISP button.

1. Check the version of Simplicity Commander. The version should be at least 1v16p8b1613 to support SiWx917 operations.

```
commander -v
```

```
Simplicity Commander 1v16p8b1613

JLink DLL version: 7.94e
Qt 5.12.10 Copyright (C) 2017 The Qt Company Ltd.
EMDLL Version: 0v19p5b744
mbed TLS version: 2.16.6

DONE
```

2. Update the board controller firmware to 1v5p0b240, or higher

   Please see the “Adapter Firmware” section of https://docs.silabs.com/simplicity-studio-5-users-guide/5.3.0/ss-5-users-guide-about-the-launcher/welcome-and-device-tabs for specific instructions, as needed.

3. Initialize security features:

   Put the device in ISP mode. On Silicon Labs’ development kits, hold down the ISP button on the radio board, then press and release the reset button on the WPK board, then release the ISP button on the radio board. Next, enter the following command

```
commander manufacturing init --mbr default
```

```
Using default MBR for SiWG917M111MGTBA...
Loading RAM code (321776 bytes)...
Starting RAM code...
Calibration code initialized.
Activation code generated successfully
DONE
```

   Note: Only execute this command once per device.

4. Create a JSON file with the keys data:

```
commander util genkeyconfig --outfile keys\keys.json --device Si917
```

```
Generating symmetric key...
Generating symmetric key...
Generating ECC P256 key pair...
Generating ECC P256 key pair...
Generating ECC P256 key pair...
Key configuration written to keys\keys.json
DONE
```

5. Save the keys to PEM format as follows:

```
commander util extractkeys keys\keys.json --dir keys
```

```
Writing ATTESTATION_PRIVATE_KEY to 'keys/attestation_private_key.pem'
Writing ATTESTATION_PUBLIC_KEY to 'keys/attestation_public_key.der'
Writing M4_OTA_KEY to 'keys/m4_ota_key.txt'
Writing M4_PRIVATE_KEY to 'keys/m4_private_key.pem'
Writing M4_PUBLIC_KEY to 'keys/m4_public_key.der'
Writing OTA_KEY to 'keys/ota_key.txt'
Writing TA_PRIVATE_KEY to 'keys/ta_private_key.pem'
Writing TA_PUBLIC_KEY to 'keys/ta_public_key.der'
DONE
```

6. Provision keys to the device. Ensure that the device is in ISP mode, according to the instructions at the beginning of this section, and enter the following command:

```
commander mfg917 provision --keys keys\keys.json
```

Note: 'commander mfg917 …' is a shorthand alias for 'commander manufacturing …'

```
Reading MBR from the connected device...
Reading 496 bytes from 0x04000000
Reading keys from provided JSON file...
Reading 1024 bytes from 0x040003e0
Reading 496 bytes from 0x04000000
Reading 112 bytes from 0x040001f0
Reading 88 bytes from 0x04000300
Reading 1024 bytes from 0x040003e0
Reading 58 bytes from 0x04000561
Reading 200 bytes from 0x04000238
Reading 72 bytes from 0x040001f0
Loading RAM code (321776 bytes)...
Starting RAM code...
Calibration code initialized.
Intrinsic keys generated successfully.
Programming TA OTA Key...
Key successfully stored
Programming M4 OTA Key...
Key successfully stored
Programming TA Public Key...
Key successfully stored
Programming M4 Public Key...
Key successfully stored
Programming attestation Key...
Key successfully stored
Programming ROM patch...
Reading 200 bytes from 0x04000238
ROM patch is already present on the device.
Programming TA MBR...
Reading 496 bytes from 0x04000000
The on-device TA MBR is already up to date.
Programming M4 MBR...
Data loaded successfully
Programming key descriptor table...
Data loaded successfully
DONE
```

7. Enable secure boot with signature validation and rollback protection by creating an mbr json file as below.

```
{
    "valids": 0,
    "puf_activation_code_addr": 8192,
    "valids": 0,
 "efuse_data": {
        "disable_m4ss_kh_access": 0,
        "m4_digital_signature_validation": 1,
        "m4_encrypt_firmware": 0,
        "m4_fw_encryption_mode": 0,
        "m4_secure_boot_enable": 1,
        "m4_anti_roll_back":1,
        "ta_digital_signature_validation": 1,
        "ta_encrypt_firmware": 0,
        "disable_m4_access_frm_tass_sec": 1,
        "ta_secure_boot_enable": 1,
        "disable_ta_jtag":0,
        "disable_m4_jtag":0
    },
    "key_desc_table_addr": 768
}
```

8. Place the board in ISP mode as described previously and provision the new MBR settings.

```
commander mfg917 provision --data mbr_security_an1442.json
```

```
Reading MBR from the connected device...
Updating MBR fields...
Reading JSON...
Loading RAM code (321776 bytes)...
Starting RAM code...
Programming TA MBR...
Data loaded successfully
Programming ROM patch...
ROM patch is already present on the device.
Programming M4 MBR...
Data loaded successfully
Programming IPMU data...
The on-device M4 IPMU data is already up to date.
DONE
```

5.2 Signing an Application for Secure boot using Simplicity Commander

This section provides a method for signing an RPS image for secure boot.

1. Build a sample app, such as the “SL Blinky” demo found in the WiSeConnect SDK available in the Simplicity Studio IDE.
2. Convert the RPS file created in the previous step to add the signature and MIC.

```
commander rps convert sl_blinky_secure.rps --sign keys\keys.json --sha SHA-256 --mic keys\keys.json --app
sl_si91x_blinky.rps
```

```
WARNING: App image size (39432 B) is not aligned to 16 B (128 bits), which is a requirement for image
encryption and MIC calculation. Appending zero-bytes to align...
Parsing MIC key 'keys.json'...
Calculating MIC of image...
Parsing signing key 'keys.json'...
Signing image...
Image SHA256: 2b4d974a43af3c3864431b4ee5c6a2cd23a8b086ba99689caf88ac470f073458
R = 2EB730D634471932F0AB72D7FE9719DAB6A91A270B31B9B588020B4C0D14B0AF
S = 21E2E1A5BD51F8181886E47465E51F7C0BB284AFC6E674D278BD0C3128700B57
RPS file successfully created at 'sl_blinky_secure.rps'.
DONE
```

3. Optionally, display the RPS file information to confirm that the file was created as intended.

```
commander util rpsinfo m4_signed_oob_demo.rps
```

```
RPS application image

Application info:
Combined image bit set  : No
Image type              : M4 application
Image size              : 0x00018D18 (101656) B
Flash address           : 0x00201000
Firmware version        : 0x00000001
Firmware version ext.   : 0x00000000
PSRAM                   : No

Security settings:
Integrity check         : MIC
MIC                     : 26 07 4E C6 6F 62 A5 3B 68 90 74 7A DE B7 81 D3
Encrypted               : No
Signed                  : Yes
SHA type                : SHA-256
Signature               :
3045022019B1666F7161B5529063391B7BA201B16DB29C8AB39B26E68B88928CAC99A259022100F371B2DED144E7CF65
845834BC48BA406114D9615C2372A00CB6A7D72338FF8A00
Boot descriptor info:
Boot desc. offset       :     0x0000
IVT offset              : 0x08202000

1 boot descriptor entry found:
Length      :   0x000000 (0)
Destination : 0x00000000

DONE
```

4. Flash the RPS file to the target board.

```
commander
        rps load m4_signed_oob_demo.rps
```

```
Uploading flashloader...
Waiting for flashloader to become ready
Writing data...
Waiting for bootloader to perform upgrade...
Resetting
DONE
```

5.3 Using and Testing Anti-rollback Protection

This example covers the steps to perform an application update, using anti-rollback prevention to prevent older firmware from being installed.

1. Ensure that anti-rollback protection is enabled through setting the m4_anti_roll_back eFuse in your target device. To accomplish this, follow steps 1-6 in Example 1, Section 5.1 Provisioning for secure boot using Simplicity Commander.
2. Create an RPS file with the firmware version set to 0. The firmware version is set to 0by default so you can use the RPS file created in step 1 in the previous example ‘Signing an application for secure boot’, Section 5.2 Signing an Application for Secure boot using Simplicity Commander.
3. Load the application to your target device, if you haven’t already done so. The command is shown in step 4 ofSection 5.2 Signing an Application for Secure boot using Simplicity Commander.
4. Increase the security version of the application using the –app-version option as shown below. Refer to Table 2.3 Firmware Version Layout in RPS Files on page 9 for the layout of the firmware version, as necessary.

```
commander rps convert m4_out_of_box_demo_security_version1.rps --app "out_of_box_demo.rps" --sha
SHA-256 --sign json\m4_private_key.pem --mic json\keys.json --app-version 0x00000100
```

```
WARNING: App image size (101636 B) is not aligned to 16 B (128 bits), which is a requirement for image
encryption and MIC calculation. Appending zero-bytes to align...
Parsing MIC key 'json\m4_ota_key.h'...
Calculating MIC of image...
Parsing signing key 'json\m4_private_key.pem'...
Signing image...
Image SHA256: cc192cff152919cdc085a5226c39435f6278fd89589beff03cbad8227fdb3725
R = BE4390FF2A1778DF48FA3C7523568ED7DBA61C9251943D0FC031C09372BE8694
S = 1F4B4502DA2D183C8601DFDD26FA16B8F7922CB18F6A8901390BCB5C6D156531
RPS file successfully created at 'm4_out_of_box_demo_security_version1.rps'.
DONE
```

5. Flash the new RPS image to the target device, the image should load and boot normally.

```
commander rps load m4_out_of_box_demo_security_version1.rps
```

```
Uploading flashloader...
Waiting for flashloader to become ready
Writing data...
Waiting for bootloader to perform upgrade...
Resetting
DONE
```

6. Optional step – confirm that the security version has been stored in OTP memory

```
commander mfg917 read efuse --out efuse.json
```

```
Reading data from region: efuse
Reading 1024 bytes from 0x40012000
Writing JSON...
Manufacturing data saved to file 'efuse.json'
DONE
```

Note: In eFuse.json, the security version is stored in big-endian format.

```
"anti_rollback_bitmap_m4_image": "0100000000000000",
```

7. Confirm that anti-rollback protection is working by attempting to load the previous image with security version 0 as follows:

```
commander rps load m4_signed_m4key_oob_demo.rps
```

```
Uploading flashloader...
Waiting for flashloader to become ready
Writing data...
Waiting for bootloader to perform upgrade...
ERROR: Waiting for flashloader failed: 108
DONE
```

Note: The image is rejected due to the anti-rollback check failing.

8. Reset the device to resume running the previously loaded application.

5.4 Moving to Production

This section describes the steps recommended for moving to production. The main difference between development and production is that during development, the master boot record (MBR) is used to configure security settings so that they can be changed if needed and the JTAG ports are enabled, while in production, the eFuses (OTP) are used to make the security settings immutable and the JTAG ports are locked.

1. Read the eFuses from a factory new device.

```
commander mfg917 read efuse --out efuse.json --device SIWG917M111MGTBA
```

```
Reading data from region: efuse
Reading 1024 bytes from 0x40012000
Writing JSON...
Manufacturing data saved to file 'efuse.json'
DONE
```

2. Open ‘eFuse.json’ in a text editor and make the following changes to enable secure boot with anti-rollback protection and signature verification for both cores. The ‘magic_byte’ field is used to indicate that the contents of the eFuse block is valid.

```
"bootloader_config": {
           "m4_anti_roll_back": 1,
           "m4_digital_signature_validation": 1,
           "safe_upgrade_frm_host": 0,
           "sdio_combo_mode": 0,
           "ta_anti_roll_back": 1,
           "ta_digital_signature_validation": 1,
           "ta_secure_boot_enable": 1,
          },
      "m4_and_security_config": {
        "common_flash_enabled": 0,
        "m4_encrypt_firmware": 0,
        "m4_flash_pinset": 0,
        "m4_flash_present": 0,
        "m4_flash_type": 0,
        "m4_secure_boot_enable": 1
    }
```

3. As a final step in production, disable the JTAG ports by making the following changes to ‘eFuse.json’, it is safe to skip this step for now if you need frequent access to the debug ports.

```
"mcu_chip_modes": {
        "disable_analog_periph": 0,
        "disable_can_interface": 0,
        "disable_cci": 0,
        "disable_ethernet": 0,
        "disable_fim_cop": 0,
        "disable_m4": 0,
        "disable_m4_access_frm_tass_sec": 0,
        "disable_m4_jtag": 1,
        "disable_m4_ulp_mode": 0,
        "disable_m4ss_kh_access": 0,
        "disable_tass_kh_access": 0,
        "disable_touch": 0,
        "disable_usb": 0,
        "disable_vad": 0,
        "disable_wurx": 0,
        "limit_m4_freq_110mhz": 0,
        "m4_flash_size": 0,
        "m4_otp_lock": 0
    },
```

4. Save the file ‘eFuse.json’

5. Write the new eFusesettings back to the target device as follows:

```
commander mfg917 write efuse --data efuse.json -d SIWG917M111MBTGA
```

```
continue
```

```
Reading JSON...
The provided data can be applied to the current Efuse.
Determining which OTP words need updating...

================================================================================
THIS IS A ONE-TIME command which PERMANENTLY writes data to the Efuse region
of your device. This command CANNOT be undone.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
```

6. Reset the target device for the settings to take effect.

```
Writing 16 bytes to OTP...
Loading RAM code (321776 bytes)...
Starting RAM code...
Efuse was successfully written to the device's OTP.
DONE
```

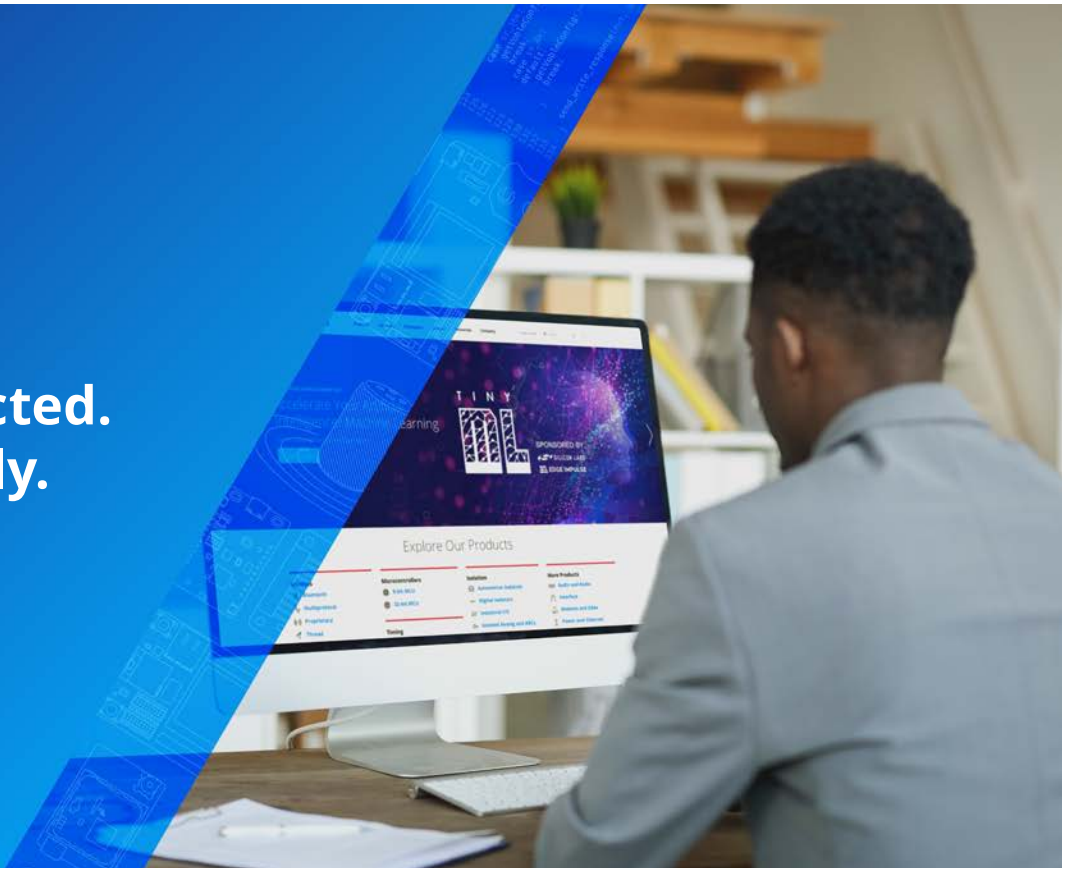
6. Revision History

Revision 0.1

October, 2024

Initial Release.

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