AN1504: Series 3 Security Overview

This application note provides a high level overview of the security features included in Series 3 devices.

Security features on Series 3 devices described in this document are implemented through the Secure Engine.

KEY POINTS

- Basic overview of Series 3 security features
- Authenticated execute in-place (AXiP)
- Mailbox/Debug challenge interface
- Automatic tamper responses
- Secure debug lock/unlock
- Secure boot/RTSL
- Secure key storage
- Secure identity
- Cryptographic hardware acceleration
- DPA/DFA countermeasures

1 Series 3 Device Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options to help developers build secure devices, secure application software, and secure paths of communication to manage those devices. Silicon Labs' security offerings were significantly enhanced by the introduction of the Series 2 products that included a Secure Engine. Series 3 products continue to include and expand upon existing Secure Engine technology. The Secure Engine is a tamper-resistant component used to securely store sensitive data and keys and to execute cryptographic functions and secure services.

1.1 User Assistance

In support of these products, we offer the following essential documentation:

Document	Summary	Applicability
AN1190: Series 2 Secure Debug	How to lock and unlock Series 2 debug access, including background information about the SE	Series 3
AN1218: Series 2 Secure Boot with RTSL	Describes the secure boot process on Series 2 devices using SE	Series 3
AN1222: Production Programming of Series 2 Devices	How to program, provision, and configure security information using SE during device production	Series 3
AN1247: Anti-Tamper Protection Configuration and Use (this document)	How to program, provision, and configure the anti-tamper module	Series 3
AN1268: Authenticating Silicon Labs Devices using Device Certificates	How to authenticate a device using secure device certificates and signatures, at any time during the life of the product	Series 3
AN1271: Secure Key Storage	How to securely "wrap" keys so they can be stored in non- volatile storage.	Series 3

Note: Documents in the above table were written for Series 2 devices and will be updated for Series 3.

1.2 Key Reference

Silicon Labs security implementations use asymmetric key pairs and symmetric keys. The table below clarifies key names, applicability, and relevant documentation.

Key Name	Customer Programmed	Purpose	Used in
Public Sign key (Public Sign Key)	Yes	Secure Boot binary authentication and/or OTA upgrade payload authentication	AN1218 (primary), AN1222
Public Command key (Public Command Key)	Yes	Secure Debug Unlock or Disable Tamper command authentication	AN1190 (primary), AN1222, AN1247
OTA Decryption key (GBL Decryption key)	Yes	Decrypting GBL payloads used for firmware up- grades	AN1222 (primary), UG266/UG489
Attestation key (Private Device Key)	No	Device authentication for secure identity	AN1268
Authenticated eXecute in Place (AXiP) Key	No	Authentication and encryption/decryption key for AXiP	AN1509
Encrypted eXecute in Place (EXiP) key	No	Encryption/Decryption key for EXIP	AN1509

1.3 SE Firmware

We strongly recommend installing the latest SE firmware on Series 3 devices to support the required security features. Refer to Example 3.6 for the procedure to upgrade the SE firmware and UG103.05 for the latest SE Firmware shipped with Series 3 devices and modules.

1.4 Acronyms Used in this Document

Acronym	Meaning
AXiP	Authenticated execute in-Place
DCI	Debug Challenge Interface
DFA	Differential Fault Analysis
DPA	Differential Power Analysis
ECC	Eliiptic Curve Cryptography
OTP	One-time programmable
SE	Secure Engine

2 Introduction

Attacks include both logical and physical attacks. Logical attacks attempt to exploit software vulnerabilities over some type of communication interface such as a wireless protocol or serial communication interface. Physical attacks require an attacker to have physical access to the device so that they perform attacks such as fault injections and side channel analysis. Defending against these attacks requires implementing a variety of security features. These include:

- Securing the application firmware using secure boot
- Locking access to the debug port using secure debug
- Mitigating fault injections using automatic tamper detection and response
- Mitigating side channel attacks using dpa, dfa against cryptographic operations
- Secure key storage

2.1 Secure Engine Principles

The following is a list of the various security features found in Series 3. This document is intended to provide an overview of each security feature. Refer to the linked documents in each section for details, including examples for implementing each feature.

2.1.1 Users and Modes

The SE has a single user – the product developer. The interfaces to the SE are described in the following section. The SE has two modes, active and inactive. When the Secure Engine is processing a command, it is active, otherwise it is inactive.

2.2 Mailbox/DCI Interface

The Secure Engine is accessible to a single user through commands that can be issued using:

- SE Mailbox (from host core) via the SE Manager API.
- Debug Challenge Interface (DCI). The Silicon Labs Simplicity Commander tool is the recommended method for using the DCI interface. The Simplicity Commander user guide is available here. The recommended version of Simplicity Commander for the steps in this application note is v1.17.4, or higher.

These commands provide access to the following features:

- Key provisioning
- Debug port configuration and locking
- Secure boot configuration and status reporting
- Tamper response configuration
- Secure engine (SE) and host firmware updates
- Code/data region configuration for EXiP/AXiP
- Cryptographic operations

2.2.1 Secure Engine Manager

The Secure Engine manager provides an interface for users to control all the features mentioned here. Documentation for SE Manager APIs can be found here. At the time of publication, this document refers to SE Manager API documentation version v5.2.0.

2.3 Authenticated eXecute in Place (AXiP)

The AXiP feature is used to encrypt and authenticate external flash memory to protect the confidentiality and integrity of code and data.

The AXiP feature is configurable in up to 8 regions, and each region can be either unprotected, encrypted (eXiP), or authenticated and encrypted (AXiP). Each region must be a multiple of 32 KB.

The configuration can be read or changed using either DCI commands or SE manager APIs. The commands are summarized in table 2.1. The default configuration is to configure region 0 for 32 KB with AXiP enabled and region 1 for the remainder of flash with AXiP enabled. Unused regions are set to size 0 and start address immediately following the previous region.

2.3.1 Users and User-controlled Security Parameters

The user of this feature is the end product developer who configures the feature through the use of the DCI interface or the SE manager APIs mentioned in table 2.2.

Table 2.1. AXiP Security Parameters

Parameter	Configuration Options	Description
Region Start Address	Start address - Integer	The starting address of one of up to 8 code regions
Region Size	Region size - integer, multiple of 32 KB	The size of the current region [1]
Region Protection Type	Integer indicating the protection type for the region	The protection mode for the code region. One of the following settings is valid. No encryption, encryption only, authentication with encryption
Code Region Closed	Boolean	Closing a code region finalizes the message authentication code (MAC) for that region.

1. Unused regions default to a size of 0.

Table 2.2. SE Manager APIs for AXiP

DCI Command(1,2)	SE Manager API	Usage
Read Region Config	sl_se_code_region_get_config	Reads the memory region configuration and reports the region sizes and protection types
Close Region	sl_se_code_region_close	Finalizes the message authentication tag (MAC) for the memory region
Apply Region Config	sl_se_code_region_apply_config	Writes the memory region configuration to set the size and protection type for each region. Some regions can have a size of 0.
Erase Region	sl_se_code_region_erase	Erases a code region
Get Region Version	sl_se_code_region_get_version	Gets the version of a code region (32bit int)
Write Region	sl_se_code_region_write	Write code region
Write Region	sl_se_data_region_write	Write to a data region
Erase Region	sl_se_data_region_erase	Erase data sectors
Get Data Region Location	sl_se_data_region_get_location	Get data region location
-	sl_se_flash_pause	Resume the previously paused long-running flash operation and/or re-enable long-running operations.
-	sl_se_flash_resume	Resume the previously paused long-running flash operation and/or re-enable long-running operations.
-	sl_se_code_region_set_active_banked	Check if a long-running operation (page/region erase, bulk write) is currently being executed on the flash
-	sl_se_flash_is_busy	Pause the ongoing long-running flash operation (and/or inhibit new long-running operations from starting), which frees up the auto-port such that code can run again (unless it tries to access the sector/region being erased, in which case the SE will need to auto-resume).

Note:

1. Performing these commands over DCI is implemented in Simplicity Commander.
2. Not all commands exist as both APIs and DCI commands.

2.4 Secure Attestation

The secure attestation feature is used to authenticate a device as a genuine Silicon Labs product. Each device is provisioned with one certificate to identify the device and another certificate to identify the batch to which the device belongs. Together with Silicon Labs' root certificate and factory certificate, a chain of trust is formed which can be used to verify the identity of a device, confirming it to be a genuine Silicon Labs product. The private key associated with the public key in the device certificate is securely stored on the device to facilitate signing of random number challenges. Refer to AN1268 for more details.

Table 2.3. SE Manager APIs for Secure Attestation

DCI Command (1,2)	SE Manager API	Usage
Vault Device Attestation	sl_se_attestation_get_psa_iat_token	Get the PSA attestation token from the SE with the given nonce.
Read Device Certificate	sl_se_read_cert	Read stored certificates (DER format) in the device.
Read Public Key from Device	sl_se_read_pubkey	Read stored Public Device Key in the device.
-	sl_se_attestation_get_config_token	Get the security configuration token from the SE with the given nonce.
-	sl_se_attestation_get_psa_iat_token_size	Get the size of a PSA attestation token with the given nonce.
-	sl_se_attestation_get_config_token_size	Get the size of a security configuration token with the given nonce.
-	sl_se_read_cert_size	Read the size of stored certificates in the device.

Note:

1. Performing these commands over DCI is implemented in Simplicity Commander.
2. Not all commands exist as both APIs and DCI commands.

2.4.1 Secure Attestation Users and User-controlled Security Parameters

Table 2.4. Secure Attestation Security Parameters

Parameter	Configuration options	Description
SE Device Identity Certificate	None[1]	X.509 certificate identifying the specific device
SE Batch Certificate	None[1]	X.509 certificate identifying the batch of devices
SE Device Private Signing Key	None[1]	ECC P-256 private key used to sign attestation tokens

1 – This parameter is provisioned in the Silicon Labs production facility and cannot be changed by the user of the product.

2.5 Secure Debug

Access to the host core's debug port can be securely locked and unlocked. The command public key is stored securely in SE OTP. The associated command private key signs a certificate that includes the device serial number, permitted debug modes, and a public key referred to as the certificate public key. The certificate private key is used to sign an access token consisting of the access command, a debug mode request, and a nonce. The random number challenge is issued by the device and must be correctly signed for the debug port to be unlocked. The debug port is automatically locked again upon reset. The random challenge can be rolled to a new value to prevent previous access tokens from being reused by unauthorized parties.

For a detailed description, see AN1190.

2.5.1 Secure Debug Users and User-controlled Security Parameters

The user of this feature is the end product developer who configures the feature through the DCI interface or the SE manager APIs mentioned in Table 2.6.

Table 2.5. Secure Debug Security Parameters

Parameter	Configuration Options	Description
Secure Debug Unlock	Boolean – enable/disable	Enables or disables the ability to securely unlock the debug port using a signed access token
Public Command Key	ECC public key	User generated ECC P-256 public key used to verify a signed certificate
Device Erase Command	Boolean – enable/disable	One-time programmable option to disable the device erase command. Disabling this command makes it impossible to reset the secure debug unlock bit or to unlock the debug port by erasing the device.
Debug lock	Boolean – enable/disable	This parameter controls the lock state of the debug port.
Non-secure Zone Invasive Debug Restriction	Boolean – enable/disable	This option enables or disables the ability to perform invasive debugging on the non-secure memory region. Only applies when using TrustZone.
Non-secure Zone Non-Invasive Debug Restriction	Boolean – enable/disable	This option enables or disables the ability to perform non-invasive debugging on the non-secure memory region. Only applies when using TrustZone.
Secure Zone Invasive Debug Restriction	Boolean – enable/disable	This option enables or disables the ability to perform invasive debugging on the secure memory region. Only applies when using TrustZone.
Secure Zone Non-Invasive Debug Restriction	Boolean – enable/disable	This option enables or disables the ability to perform non-invasive debugging on the secure memory region. Only applies when using TrustZone.
Device Serial Number	Not configurable by the user	128-bit unique identifier provisioned by Silicon Labs production process.

Table 2.6. SE Manager APIs for Secure Debug

DCI Command (1)	SE Manager API	Usage
Enable Secure Debug	sl_se_enable_secure_debug	Enables the ability to securely unlock the debug port using a signed access token.
Disable Secure Debug	sl_se_disable_secure_debug	Disables the ability to securely unlock the debug port using a signed access token.
Set Debug Options	sl_se_set_debug_options	Sets options for restricting invasive/non-invasive debug access in the secure/non-secure memory regions.
Disable Device Erase	sl_se_disable_device_erase	Disables the device erase SE command. Once enabled, it is impossible to unlock the device through an erase.
Apply Lock	sl_se_apply_debug_lock	Locks the debug port. Unlocking depends on the secure debug lock and disable device erase settings.
- (2)	sl_se_get_challenge	Gets a random number challenge from the SE to be signed.
Roll Challenge	sl_se_roll_challenge	Rolls the random number challenge for unlocking the debug port.
Secure Debug Unlock	sl_se_open_debug	Unlocks the debug port. Requires a signed certificate.
Read Lock Status	sl_se_get_debug_lock_status	Returns the current state of the debug port lock.
Disable Device Erase	sl_se_disable_device_erase	Disables the Erase Device command. This command does not lock the debug interface to the part, but it is an IRREVERSIBLE action for the part.

Note:

1. Performing these commands over DCI is implemented in Simplicity Commander.
2. Device challenge is retrieved through the secure unlock command.

2.6 Secure Key Storage

The SE isolates cryptographic functions and data from the host core. The SE is used to accelerate cryptographic operations as well as provide a method to securely store keys.
The SE contains OTP key storage slots for three specific keys:
1. The Public Sign Key, used for Secure Boot and Secure Upgrades
2. The Public Command Key, used for Secure Debug unlock and tamper disable
3. The Symmetric OTA Decryption Key, used for Over-The-Air updates
These keys are one-time programmable and are persistent for the lifetime of the device.

2.6.1 Secure Key Storage Users and User-controlled Security Parameters

The user of this feature is the end product developer who configures the feature using the SE manager APIs mentioned in table 2.8 and DCI commands through Simplicity Commander.

Table 2.7. Security Parameters for Secure Key Storage

Parameter	Configuration Options	Description
Key Size	Integer – number of bits	Indicates the size of the key in bits
Key Type	Integer – type of key	Indicates the type of key symmetric or asymmetric
Key Permissions	Bitfield – allowed uses for the key	A set of flags indicating the permitted uses for the key, the exportability of the key, and whether the key is in volatile or non-volatile memory
Key Storage	Data structure pointing to the physical storage location of the key	A pointer to the physical location of the key material together with a size parameter indicating the size of the key material
Key Password	Data structure	An optional 8-byte value which, if set, is required to use the key
Key Domain	Data structure	For ECC asymmetric keys, a set of data that indicates the parameters for the curve used

2.6.2 SE Volatile Storage

The SE also contains four volatile storage slots for other user keys. These slots are not persistent through a reset. In the case where a key needs persistent storage, the key must be stored wrapped outside of the SE in non-volatile storage. After a device reset, the key can be loaded into the SE volatile key storage for usage by index or used in-place (passed to the SE on every requested operation).

2.6.3 Wrapped Keys

Without any secure key storage mechanism, the user key stored in non-volatile storage is opened to storage-extraction attacks (such as gaining access to and downloading device flash), as well as application-level attacks (i.e., taking control of the user application or privileges in a manner that allows access to the keys).
With Secure Key Storage, a user can only access a key from the SE in a “wrapped” format. In this format, the key is encrypted by a device-unique root key, only available to the SE. This allows a user to store a key confidentially in non-volatile storage to provide key persistence. Using Secure Key Storage, the plaintext key is never stored in non-volatile memory, preventing storage-extraction attacks from obtaining the key. After a device reset, the wrapped key can be loaded from non-volatile storage into the SE. The SE then unwraps the key and stores it in one of the key storage slots without ever exposing the plaintext key to the application, which also prevents application-level attacks from exposing the key. To perform cryptographic operations using a wrapped key, the key is referred to by its identifier. Refer to AN1271 for more details.

2.6.4 Key Slot Unit (KSU)

The KSU module provides local key storage for wrapped keys. The SE can unwrap a wrapped key and place it in the KSU for the appropriate AES engine. Each key contains metadata that controls which AES engine may use it. The host cannot read the contents of the KSU slots directly, it can only reference the keys using their slot identifiers. KSU slots are volatile, meaning their contents are not visible when the device is powered off.

Note: DFA countermeasures are only available in the SE cryptographic accelerator, not in the host cryptographic accelerators. Any operations that should be protected against DFA attacks must be restricted to the SE.

Table 2.8. SE Manager APIs for Secure Key Storage

SE Manager API	Usage
sl_se_generate_key	Generate a new key and store it either in a volatile HSE storage slot or as a wrapped key.
sl_se_import_key	Import a plaintext key and store it either in a volatile HSE storage slot or as a wrapped key.
sl_se_export_key	Export a volatile or wrapped key back to plaintext if allowed. It will fail for a key that has been flagged as SL_SE_KEY_FLAG_NON_EXPORTABLE.
sl_se_transfer_key	Transfer a volatile or wrapped key to another storage option (volatile HSE storage slot or a wrapped key) if allowed.
sl_se_delete_key	Delete a key from a volatile SE storage slot.
sl_se_generate_key	Generate a new key and store it either in a volatile HSE storage slot or as a wrapped key.
sl_se_import_key	Import a plaintext key and store it either in a volatile HSE storage slot or as a wrapped key.
sl_se_export_key	Export a volatile or wrapped key back to plaintext if allowed. It will fail for a key that has been flagged as SL_SE_KEY_FLAG_NON_EXPORTABLE.
sl_export_public_key	Export the public part of an ECC keypair.
sl_se_validate_key	Validates a key descriptor to determine if all of the required properties for that key type have been set.
sl_se_get_storage_size	Returns the required storage size for the given key.
sl_se_init_otp_key	Provision a key to SE OTP. Used for symmetric and public keys.
sl_se_read_pubkey	Read out a public key from SE OTP.

2.7 Secure Boot and Root of Trust Secure Loader (RTSL)

One of the most important criteria for a secure product is to ensure that the firmware that defines the behavior of that product is authentic. Secure boot ensures that firmware is authentic by verifying the signature of that firmware using a tamper-resistant public key. The firmware is only allowed to run once it has been verified to be authentic. Another important feature of secure boot is roll-back protection. Firmware rollbacks should be prevented to mitigate an attacker's ability to take advantage of vulnerabilities in previous firmware versions. Refer to AN1218 for more details.

2.7.1 Secure Boot Users and User-controlled Security Parameters

The user of this feature is the end product developer who configures the feature through the use of the SE manager APIs and DCI commands mentioned in table 2.6.

Table 2.9. Secure Boot Security Parameters

Parameter	Configuration options	Description
Secure Boot Enable	Boolean – enable/disable	OTP flag to indicate whether secure boot is enforced or not
Secure Boot Anti-Rollback	Boolean – enable/disable	OTP flag to indicate whether rollback protection of host firmware is enabled or not
Secure Boot Verify Certificate	Boolean – enable/disable	OTP flag to indicate whether certificate-based signing of host firmware is required or not
Secure Boot Public Sign Key	ECC P-256 public key	User-generated ECC P-256 public key. Used to verify either the host firmware certificate when certificate-based secure boot is used, or the host firmware signature itself when certificate-based secure boot is not used
Secure Boot Certificate	Proprietary data structure	A certificate containing a version and a public key used to verify host firmware signature. This is injected into the host firmware image binary
Host Firmware Signature	ECDSA signature	User-generated signature created with ECDSA on ECC P-256 curve with the user's private signing key. Injected into the host firmware image binary

Table 2.10. SE Manager APIs for Secure Boot

DCI Command(1,2)	SE Manager API	Usage
Write User Configuration	sl_se_init_otp	Provision the configuration settings to OTP. Settings include secure boot enforcement, anti-rollback of host firmware, flash page locking, and tamper response settings. Settings are provisioned as a single word. Therefore, all settings must be set in a single operation
Read User Configuration	sl_se_read_otp	Read out the OTP settings currently provisioned to the device
Get Device Status	sl_se_get_status	Read the boot status of the SE. The information retrieved also includes SE firmware version, state of the debug port lock, tamper status, secure debug configuration
-	sl_se_check_se_image	Validate SE firmware image
-	sl_se_apply_se_image	Apply SE firmware image
-	sl_se_get_upgrade_status_se_image	Get upgrade status of SE firmware image
-	sl_se_check_host_image	Validate Host firmware image
-	sl_se_apply_host_image	Apply Host firmware image
-	sl_se_get_upgrade_status_host_image	Get upgrade status of Host firmware image

Note:

1. Performing these commands over DCI is implemented in Simplicity Commander.
2. Not all commands exist as both APIs and DCI commands.

2.7.2 SE Boot Status

The SE can be queried through the DCI interface to determine the boot status. Refer to AN1218 section 6 for a full list of the status codes.

2.8 Cryptographic Acceleration

Security Parameters for Cryptographic Acceleration

Table 2.11. Security Parameters for Cryptographic Algorithms

Parameter	Configuration Options	Description
Private Key	Integer – size depends on algorithm	For ECDSA/EdDDSA algorithms, this is the key used for signing. For key agreement algorithms, this is the key used together with a peer's public key for establishing a shared secret.
Public Key	Derived from associated private key and elliptic curve	For ECDSA/EdDDSA algorithms, this is the key used for verification. For key agreement algorithms, this is the key used together with a peer's private key for establishing a shared secret.
Elliptic Curve	One of the following: • P-192 • P-256 • Curve25519	A set of parameters that define an elliptic curve. These are used to associate a public key with a private key.
Symmetric Key	Key material	128,192, or 256 bits of key material used for encrypting/decrypting/authentication.
Key Size	For symmetric algorithms, 128, 192, or 256 bits	The size of the key to be used in AES operations.
Algorithm	One of the following: • AES-ECB • AES-CFB8 • AES-CFB128 • AES-CBC • AES-CTR • AES-CMAC • AES-CCM • AES-GCM • ECDSA • EdDSA • ECDH • SHA1/SHA2/HMAC-AES-MMO • EC-JPAKE • HMAC	The cryptographic algorithm to be performed.
Operation	Encrypt or decrypt	This parameter controls the behavior of algorithms that support encryption or decryption through the same API function.
Nonce	Data	Initialization value for supported cipher algorithms.
Nonce Length	Integer – allowable values depend on chosen algorithm	The length of the nonce passed to the cryptographic algorithm.
Input Data	Data	The data to be processed by the cryptographic algorithm.
Associated Data	Data	For supported algorithms, data that is to be used in the generation of an authentication tag but not encrypted or decrypted.
Output Data	Storage location	The storage location for the data output by the cryptographic algorithm.
Data Length	Integer – allowable values depend on chosen algorithm	The length of the data to be operated on by the cryptographic algorithm. Configurable for all algorithms except AES-ECB.
Authentication Tag	Storage location	Location where an authentication tag such as a CCM or GCM tag is to be stored.
Authentication Tag Length	Integer – allowable values depend on chosen algorithm	The length of authentication tag to be generated or verified by the chosen cryptographic algorithm.
Password	Data	Password for EC-JPAKE.
Password Length	Integer	Length of the EC-JPAKE password in bytes.
Role	Boolean – client or server	The role for EC-JPAKE. Only values indicating client or server are allowable.
Hash Type	Integer	The type of hash to use. Supported hashes are SHA1, SHA2/256/384/512, and HMAC-AES-MMO.

Refer to SE Manager API for more information.

2.8.1 Asymmetric Cryptography

Asymmetric cryptographic operations, those that operate on public/private keypairs, are always performed by the SE core. The supported curves and operations are summarized in table 2.12.

Table 2.12. Supported Curves and Operations

Curve	Supported Operations
P-192,P-256	ECDSA,ECDH, EC-JPAKE
Curve25519	ECDH,EdDSA

All operations are available through the mailbox interface. The APIs are summarized in table 2.13.

Table 2.13. SE Manager APIs for Asymmetric Cryptographic Functions

SE Manager API	Usage
sl_se_ecc_sign	This function computes Elliptic-Curve Cryptography (ECC) digital signatures of a message.
sl_se_ecc_verify	This function verifies Elliptic-Curve Cryptography (ECC) digital signatures of a message.
sl_se_ecdh_compute_shared_secret	This function computes the shared secret with Elliptic Curve Diffie Hellman (ECDH) algorithm.
sl_se_ecjpake_check	Check if an EC J-PAKE context is ready for use.
sl_se_ecjpake_derive_secret	Derive the shared secret (TLS: Pre-Master Secret).
sl_se_ecjpake_free	This clears an EC J-PAKE context and frees any embedded data structure.
sl_se_ecjpake_init	Initialize an EC J-PAKE context.
sl_se_ecjpake_read_round_one	Read and process the first round message (TLS: contents of the Client/ServerHello extension, excluding extension type and length bytes).
sl_se_ecjpake_read_round_two	Read and process the second round message (TLS: contents of the Client/ServerKeyExchange).
sl_se_ecjpake_setup	Set up an EC J-PAKE context for use.
sl_se_ecjpake_write_round_one	Generate and write the first round message (TLS: contents of the Client/ServerHello extension, excluding extension type and length bytes).
sl_se_ecjpake_write_round_two	Generate and write the second round message (TLS: contents of the Client/ServerKeyExchange).

2.8.2 Symmetric Cryptography

Symmetric cryptographic operations such as AES and hash operations can be performed by the SE, when high performance on large blocks with high security is desired or by one of the host cryptographic accelerators. The following symmetric cipher modes are supported with key sizes of 128/192/256 bits:

- ECB
- CTR
- CBC
- CFB
- CCM
- GCM
- CBC-MAC

The following hashes are supported:

- SHA-1
- SHA-2/256/384/512
- HMAC-AES-MMO

Operations to be performed by the SE use the mailbox interface. The following user-friendly APIs are provided for these operations.

Table 2.14. SE Manager APIs for Symmetric Cryptographic Functions

SE Manager API	Usage
sl_se_aes_crypt_ecb	AES-ECB block encryption/decryption.
sl_se_aes_crypt_cbc	AES-CBC buffer encryption/decryption.
sl_se_aes_crypt_cfb128	AES-CFB128 buffer encryption/decryption.
sl_se_aes_crypt_cfb8	AES-CFB8 buffer encryption/decryption.
sl_se_aes_crypt_ctr	AES-CTR buffer encryption/decryption.
sl_se_ccm_encrypt_and_tag	AES-CCM buffer encryption.
sl_se_ccm_auth_decrypt	AES-CCM buffer decryption.
sl_se_ccm_multipart_starts	Prepare a CCM streaming command context object.
sl_se_ccm_multipart_update	This function feeds an input buffer into an ongoing CCM computation.
sl_se_ccm_multipart_finish	Finish a CCM streaming operation and return the resulting CCM tag.
sl_se_gcm_crypt_and_tag	This function performs GCM encryption or decryption of a buffer.
sl_se_gcm_auth_decrypt	This function performs a GCM authenticated decryption of a buffer.
sl_se_cmac	This function calculates the full generic CMAC on the input buffer with the provided key.
sl_se_cmac_multipart_starts	Prepare a CMAC streaming command context object.
sl_se_cmac_multipart_update	This function feeds an input buffer into an ongoing CMAC computation.
sl_se_cmac_multipart_finish	Finish a CMAC streaming operation and return the resulting CMAC tag.
sl_se_gcm_multipart_starts	Prepare a GCM streaming command context object.
sl_se_gcm_multipart_update	This function feeds an input buffer into an ongoing GCM computation.
sl_se_gcm_multipart_finish	Finish a GCM streaming operation and return the resulting GCM tag.
sl_se_hmac	Compute an HMAC on a full message.
sl_se_hmac_multipart_starts	Prepare an HMAC streaming command context object to be used in subsequent HMAC streaming function calls.
sl_se_hmac_multipart_update	This function feeds an input buffer into an ongoing HMAC computation.
sl_se_hmac_multipart_finish	Finish an HMAC streaming operation and return the resulting HMAC.

2.9 Countermeasures for Side-channel and Other Physical Attacks

Side channel attacks observe physical quantities such as power consumption to extract information such as secret keys. Physical fault injection can be used to alter program behavior. Refer to the following section for specific types of fault injection.

2.9.1 Fault Injection

2.9.1.1 Power Supply Glitching

Injecting current into the power supply of a victim device can cause applications to misbehave by altering bits in non-volatile memory. This type of attack is mitigated using a brown-out detector (BOD) which triggers a configurable automatic response.

2.9.1.2 Electromagnetic Glitching

The presence of an electromagnetic pulse can cause applications to misbehave by altering bits in memory. Glitch detection circuits detect these fault injections and trigger a configurable automatic response.

2.9.1.3 Temperature Glitching

Applications running on devices operating near their temperature limit may misbehave. A temperature sensor monitors the device temperature and can trigger a configurable automatic tamper response when the temperature is above or below the defined limits.

2.9.1.4 Enclosure Opening

Most attacks in this section require the product's enclosure to be opened as a prerequisite. This can be detected with the External Tamper Detect (ETAMPDET) module.

Refer to AN1247 for details on automatic tamper responses.

Table 2.15. SE Manager APIs for Secure Tamper Responses

DCI Command	SE Manager API	Usage
Write User Configuration	sl_se_init_otp	Initialize SE OTP configuration (including tamper configuration on HSE-SVH devices).
Read User Configuration	sl_se_read_otp	Read SE OTP configuration (including tamper configuration on HSE-SVH devices).
Generate Access Certificate	sl_se_get_serialnumber	Read out the serial number (16 bytes) of the HSE device.
Roll Challenge	sl_se_get_challenge	Read out the current challenge value (16 bytes) for tamper disable.
Get Device Status	sl_se_get_status	Read the current HSE status (including recorded tamper status on HSE-SVH devices).
-	sl_se_get_tamper_reset_cause	Read the cached value of the EMU->TAMPERRSTCAUSE register after a tamper reset.
-	sl_se_disable_tamper	Temporarily disable tamper configuration using the Disable Tamper Token.
-	sl_se_get_reset_cause	Read the EMU->RSTCAUSE register from HSE devices after a tamper reset.
-	sl_se_get_tamper_reset_cause	Read the cached value of the EMU->TAMPERRSTCAUSE register after a tamper reset.
-	sl_se_enter_active_mode	Force the SE to remain active to enable the detection of glitch tamper events on the host Cortex-M33 core.

-	sl_se_exit_active_mode	Exit active mode and allow the SE to sleep when not performing operations. This will prevent the detection of glitch tamper events when the SE is sleeping. This API should only be used if active mode was entered by calling sl_se_enter_active_mode. If active mode is set through a DCI command, it can only be disabled through a DCI command.

Note:

1. Performing these commands over DCI is implemented in Simplicity Commander.
2. Not all commands exist as both APIs and DCI commands.

2.9.2 Differential Power Analysis (DPA)

Observing the power consumption of a device may leak information. DPA attacks are mitigated by “blinding” the power consumption of the device in a randomized pattern. DPA countermeasures are always on and cannot be disabled.

2.9.3 Differential Fault Analysis (DFA)

Injecting faults during cryptographic operations may leak information about the internal state of the cryptographic accelerator. DFA attacks are mitigated by performing the same cryptographic operation more than once to compare the results. DFA countermeasures are always on and cannot be disabled.

2.9.4 Secure Tamper Response User-controlled Security Parameters

Table 2.16. Secure Tamper Response Security Parameters

Parameter	Configuration options	Description
Filter Period	Integer between 0 - 7	Determines the filter period. A tamper event is triggered if a number of filtered tamper events equal to or greater than the filter threshold occurs within the time determined by the filter period. Filter period is 32 ms *2^n
Filter Threshold	Integer between 0 – 31	Determines the number of events required to trigger a filtered tamper. Threshold = 256/2^n
Reset Threshold	Integer between 0 - 255	Determines the number of times the device is reset due to a tamper before entering debug mode. Setting to “0” means never enter debug mode.
Active During Sleep	Boolean – enable/disable	Determines whether tamper detection is active while the device is in sleep mode
Digital Glitch Always On	Boolean – enable/disable	Determines whether glitch detection is always active
Crypto Error	One of Reset or erase OTP	Tamper source automatic response
DCI Authorization	One of Ignore, filter, interrupt, reset or erase OTP	
Decouple Brownout Detect	One of Reset or erase OTP	
DPLL Fail	One of Ignore, filter, interrupt, reset or erase OTP	
External Tamper Detect	One of Ignore, filter, interrupt, reset or erase OTP	
Filter Counter	One of Ignore, filter, interrupt, reset or erase OTP	
Glitch Detector	One of Ignore, filter, interrupt, reset or erase OTP	
KSU 1-bit ECC Error	One of Ignore, filter, interrupt, reset or erase OTP	
KSU 2-bit ECC Error	One of Reset or erase OTP	
L2 ICACHE Parity Error	One of Reset or erase OTP	

Mailbox Authorization	One of Ignore, filter, interrupt, reset or erase OTP	
OTP Alarm	One of Reset or erase OTP	
PRS0	One of Ignore, filter, interrupt, reset or erase OTP	
PRS1	One of Ignore, filter, interrupt, reset or erase OTP	
PRS2	One of Ignore, filter, interrupt, reset or erase OTP	
QSPI Reseed Error		
Secure Boot Fail	One of Ignore, filter, interrupt, reset or erase OTP	
Secure Lock Error	One of Reset or erase OTP	
Selftest Error	One of Reset or erase OTP	
SE Assert	One of Reset or erase OTP	
SE CPU Major Fault	One of Reset or erase OTP	
SE ICACHE Error	One of Reset or erase OTP	
SE RAM ECC 1-bit Error	One of Ignore, filter, interrupt, reset or erase OTP	
SE RAM ECC 2-bit Error	One of Reset or erase OTP	
SE Watchdog	One of Reset or erase OTP	
SOC PLL Fail	One of Ignore, filter, interrupt, reset or erase OTP	
Temperature Sensor	One of Ignore, filter, interrupt, reset or erase OTP	
TRNG Monitor	One of Ignore, filter, interrupt, reset or erase OTP	

3 Examples

3.1 Provisioning for Security

1. Run the following command to generate keypairs for secure boot and secure debug:

```
commander util 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
```

```
commander util genkey --type ecc-p256 --privkey command_key.pem --pubkey command_pubkey.pem
```

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

2. Provision the Public Sign Key in Simplicity Commander.

To write the Public Sign Key to the device, run the command:

```
commander security writekey --sign sign_pubkey.pem --device SIMG301M104LIH --serialno 440048205
```

```
Device has serial number 000000000000000014b457fffe045a8e
================================================================================
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
```

where sign_pubkey.pem is the Public Sign Key in Privacy Enhanced Mail (PEM) format. A third party cannot change this setting once it has been set.

3. Provisioning the Public Command Key in Simplicity Commander

   To write the Public Command Key to the device, run the command:

```
commander security writekey --command command_pubkey.pem --device SiMG301M104LIH--serialno
440048205
```

```
Device has serial number 00000000000000014b457fffe045a8e
================================================================================
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
```

   where command_pubkey.pem is the Public Command Key in PEM format. A third party cannot change this setting once it has been set.

4. Provisioning the GBL Decryption Key

   To generate the text file for the GBL Decryption Key, run the command:

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

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

where aes_key.txt contains the randomly generated AES-128 key.

To write the GBL Decryption Key to the HSE device, run the command:

```
commander security writekey --decrypt aes_key.txt --device  SiMG301M104LIH--serialno 440048205
```

```
Device has serial number 00000000000000014b457fffe045a8e
================================================================================
Please look through any warnings before proceeding.
THIS IS A ONE-TIME command, any encrypting of GBL files must be done with this key.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
================================================================================
continue
DONE
```

3.2 Enabling Secure Boot and Tamper Configuration

The Secure Boot feature verifies the integrity and authenticity of the host application before allowing it to execute. Enabling this feature is IRREVERSIBLE, which means once enabled, Secure Boot can no longer be disabled throughout the life of the device. The Secure Boot settings are written to the one-time-programmable (OTP) memory. They cannot be changed once programmed. The anti-tamper configuration is provisioned with Secure Boot settings. The anti-tamper configuration determines the response from the device if a tamper event occurs.

Note:
The user_configuration.json is a JSON file that contains the desired Secure Boot settings and anti-tamper configuration. Use the following command on the target device (e.g., SiMG301M104LIH) to generate a default configuration file. The content of the JSON file is device-dependent (--device <device name>).

The security genconfig command above generates a generic configuration file for SiMG301M104LIH consisting of the properties listed in Table 3.1 Secure Boot Items (mcu_flags) for Series 3 Devices and Table 3.2 Tamper Items for Series 3 devices. A text editor can be used to modify the default settings shown below to the desired configuration.

```
commander security genconfig --nostore -o user_configuration.json --device SiMG301M104LIH --serialno 440048205
```

```
DONE
```

Table 3.1. Secure Boot Items (mcu_flags) for Series 3 Devices

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

3.3 Enabling Anti-Rollback Protection and Certificate-Based Secure Boot

Enable anti-rollback protection and certificate-based secure boot as follows:

1. Open the user_config.json file created in the previous step with any text editor.
2. Change the SECURE_BOOT_VERIFY_CERTIFICATE setting from 'false' to 'true' and save the file.

```
{
   "OPN": "SIMG301M104LIH",
   "VERSION": "1.0.0",
   "mcu_flags": {
      "SECURE_BOOT_ANTI_ROLLBACK": true,
      "SECURE_BOOT_ENABLE": true,
      "SECURE_BOOT_VERIFY_CERTIFICATE": true
   },
   "tamper_filter": {
      "FILTER_PERIOD": 0,
      "FILTER_THRESHOLD": 0,
      "RESET_THRESHOLD": 0
   },
   "tamper_flags": {
      "ALIVE_DURING_SLEEP": false,
      "DGLITCH_ALWAYS_ON": false
   },
   "tamper_levels": {
      "CRYPTO_ERROR": 4,
      "DCI_AUTH": 0,
      "DECOUPLE_BOD": 4,
      "DPLL_FAIL": 0,
      "ETAMPDET": 0,
      "FILTER_COUNTER": 0,
      "GLITCH_DETECTOR": 0,
      "KSU_1_BIT_ECC_ERROR": 0,
      "KSU_2_BIT_ECC_ERROR": 4,
      "L2ICACHE_PARITY_ERROR": 4,
      "MAILBOX_AUTH": 0,
      "OTP_ALARM": 4,
      "PRS0": 0,
      "PRS1": 0,
      "PRS2": 0,
      "QSPI_RESEED_ERROR": 4,
      "SECURE_BOOT_FAILED": 0,
      "SECURE_LOCK_ERROR": 4,
      "SELFTEST_ERROR": 4,
      "SE_ASSERT": 4,
      "SE_CPU_MAJOR_FAULT": 4,
      "SE_ICACHE_ERROR": 4,
      "SE_RAM_ECC_1BIT": 0,
      "SE_RAM_ECC_2BIT": 4,
      "SE_WATCHDOG": 4,
      "SOCPLL_FAIL": 0,
      "TEMP_SENSOR": 0,
      "TRNG_MONITOR": 0
   }
}
```

Table 3.2 Tamper Items for Series 3 Devices

Name	Description
tamper_levels	The tamper levels of different tamper sources.
tamper_filter	The settings for tamper filters.
tamper_flags	The settings for tamper flags.

3. The following command writes the Secure Boot settings and anti-tamper configuration in user_configuration.json file to the device. This command can be executed only once per device.

```
commander security writeconfig --configfile user_configuration.json --device SiMG301M104LIH --serialno 440048205
```

4. To check the device's Secure Boot settings and anti-tamper configuration, run the security readconfig command.

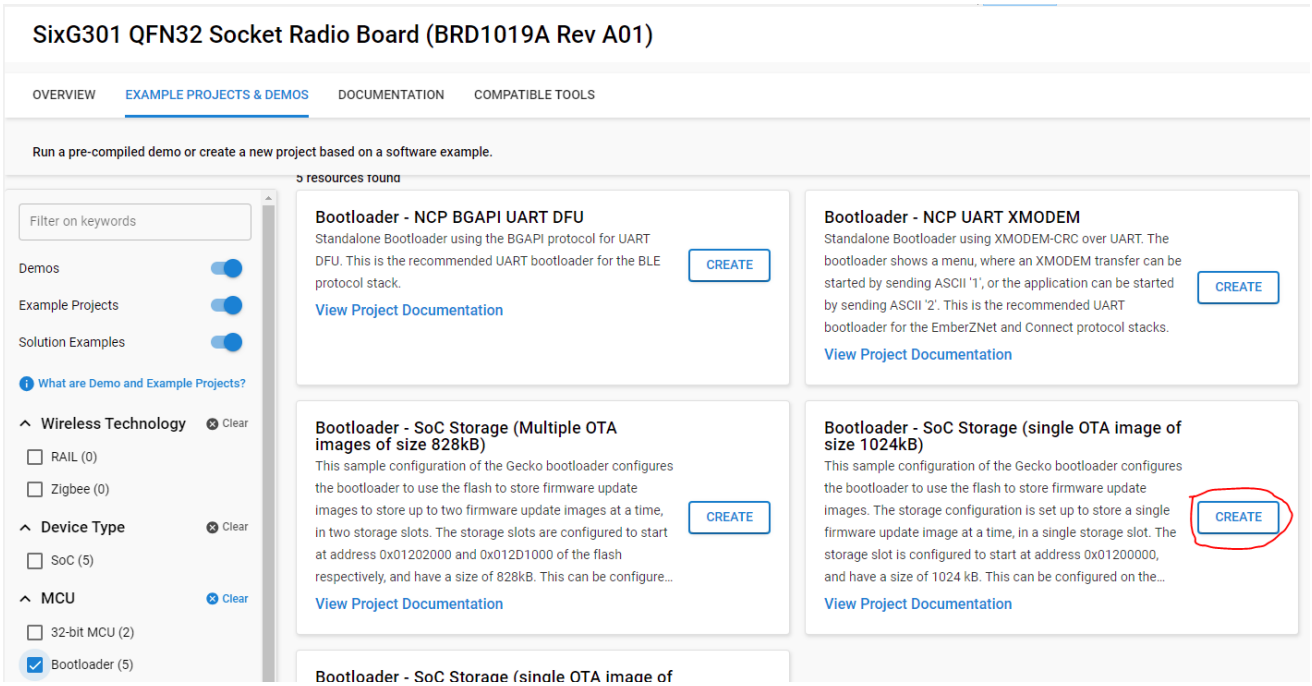
```
================================================================================
THIS IS A ONE-TIME configuration: Please inspect file before confirming:
user_configuration.json
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
================================================================================
continue
DONE
```

```
commander security readconfig --device SiMG301M104LIH  --serialno 440048205
```

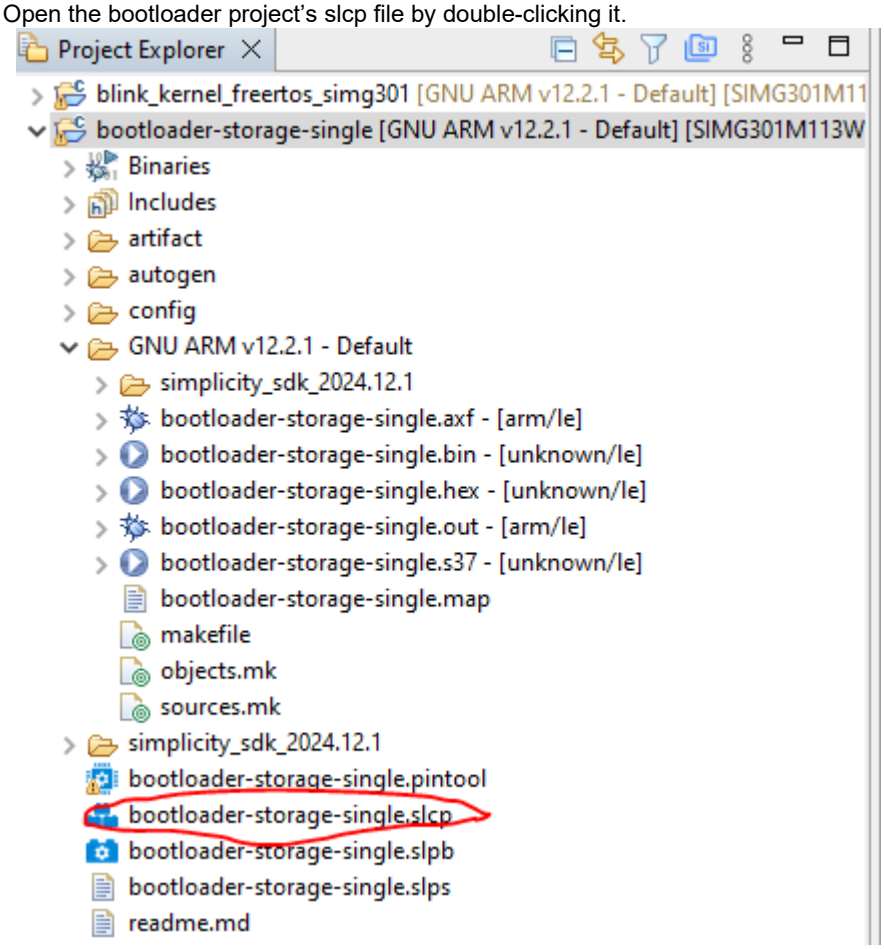
```
MCU Flags
Secure Boot                    : Enabled
Secure Boot Verify Certificate : Enabled
Secure Boot Anti Rollback      : Enabled
Secure Boot Page Lock Narrow   : Disabled
Secure Boot Page Lock Full     : Disabled

Tamper Levels
FILTER_COUNTER     : 0
SE_WATCHDOG        : 4
CRYPTO_ERROR       : 4
SE_RAM_ECC_2BIT    : 4
SE_CPU_MAJOR_FAULT : 4
L2ICACHE_PARITY_ERROR: 4
SECURE_BOOT_FAILED : 0
MAILBOX_AUTH       : 0
DCI_AUTH           : 0
SE_ASSERT          : 4
SELFTEST_ERROR     : 4
TRNG_MONITOR       : 0
SECURE_LOCK_ERROR  : 4
GLITCH_DETECTOR    : 0
OTP_ALARM          : 4
SE_ICACHE_ERROR    : 4
SE_RAM_ECC_1BIT    : 0
DECOUPLE_BOD       : 4
TEMP_SENSOR        : 0
DPLL_FAIL          : 0
SOCPLL_FAIL        : 0
ETAMPDET           : 0
KSU_1_BIT_ECC_ERROR: 0
KSU_2_BIT_ECC_ERROR: 4
QSPI_RESEED_ERROR  : 4
PRS0               : 0
PRS1               : 0
PRS2               : 0

Tamper Filter
Filter Period   : 0
Filter Threshold : 0
Reset Threshold  : 0

Tamper Flags
Digital Glitch Detector Always On: Disabled
Keep Tamper Alive During Sleep: Disabled
DONE
```

3.4 Secure Boot

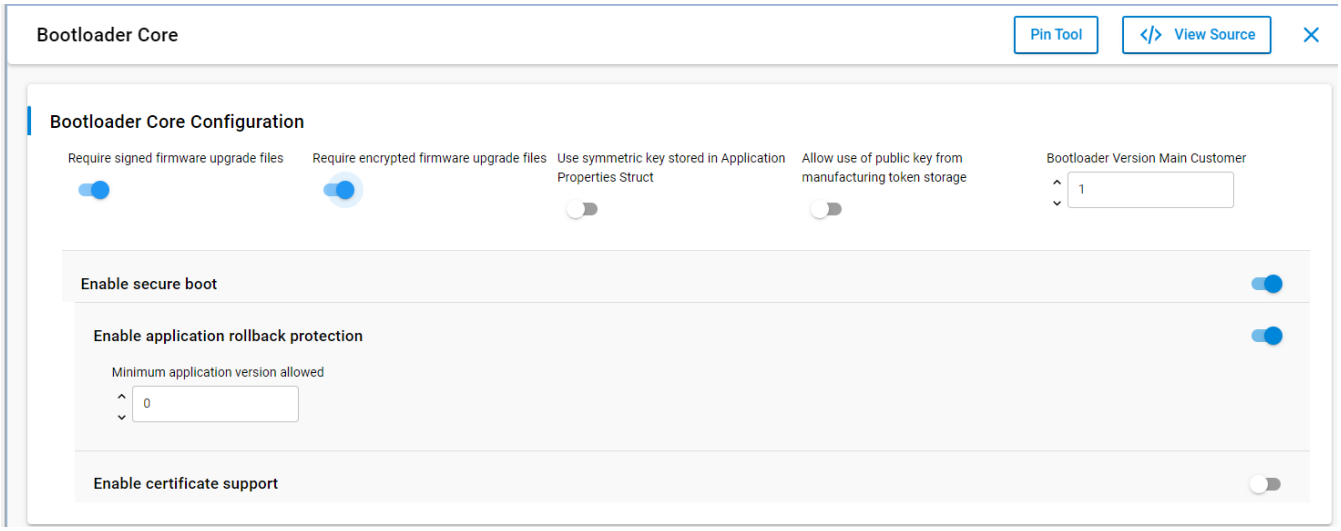
1. Create a bootloader project for your chosen device as shown below:



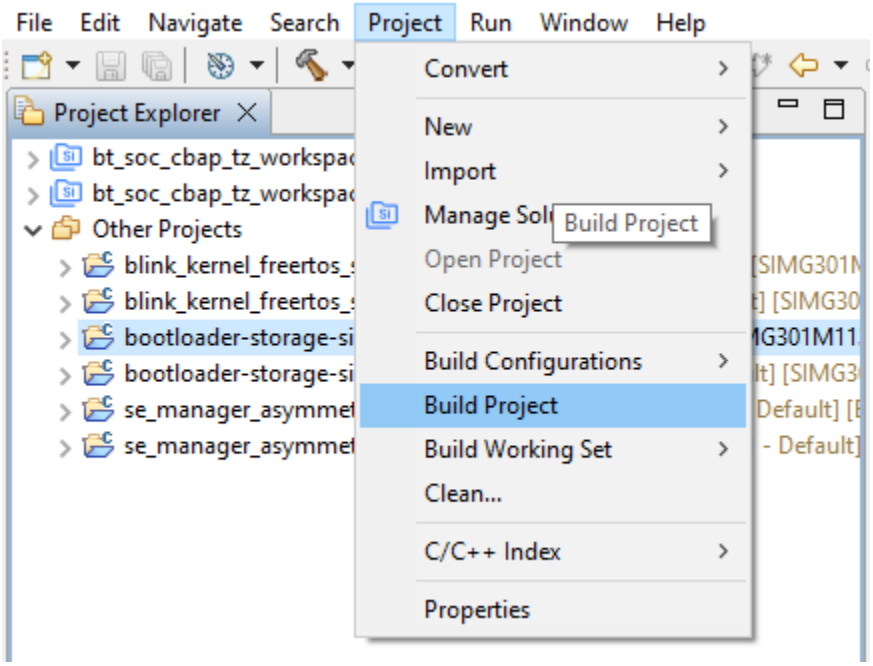
2. Open the bootloader project's slcp file by double-clicking it.



3. In the bootloader core component, enable the secure boot features as shown:



4. Build the bootloader project as shown.



5. Create a keypair for a certificate.

```
commander util genkey --type ecc-p256 --privkey cert_sign_key.pem --pubkey cert_sign_pubkey.pem
```

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

6. Create a certificate with the new key, signing the certificate with the BL private key.

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

```
Successfully signed certificate
DONE
```

7. Add the certificate into the bootloader.

```
commander convert bootloader-storage-single.s37 --secureboot --certificate bl_cert.bin --keyfile
cert_sign_key.pem --outfile bootloader-storage-single-signed-cert-v1.s37
```

```
Parsing file bootloader-storage-single.s37...
Adding certificate to image...
Private key matches public key in certificate.
Found Application Properties at 0x0100330c
Writing Application Properties signature pointer to point to 0x01003810
Setting signature type in Application Properties: 0x00000001
Image SHA256: 2e51b2b3f41b5a42f3072bc62e843a3f4406dfe4afd46de20e03b508b662acf7
R = 15209BFF03BCD42CA05C0C14175D90D2E25A5FAEDB0ADA18B511144B94C79A13
S = C37886136746A193C78AD4AADE541F373A6D61BDBC462C4D6BAD188610FDF49D
```

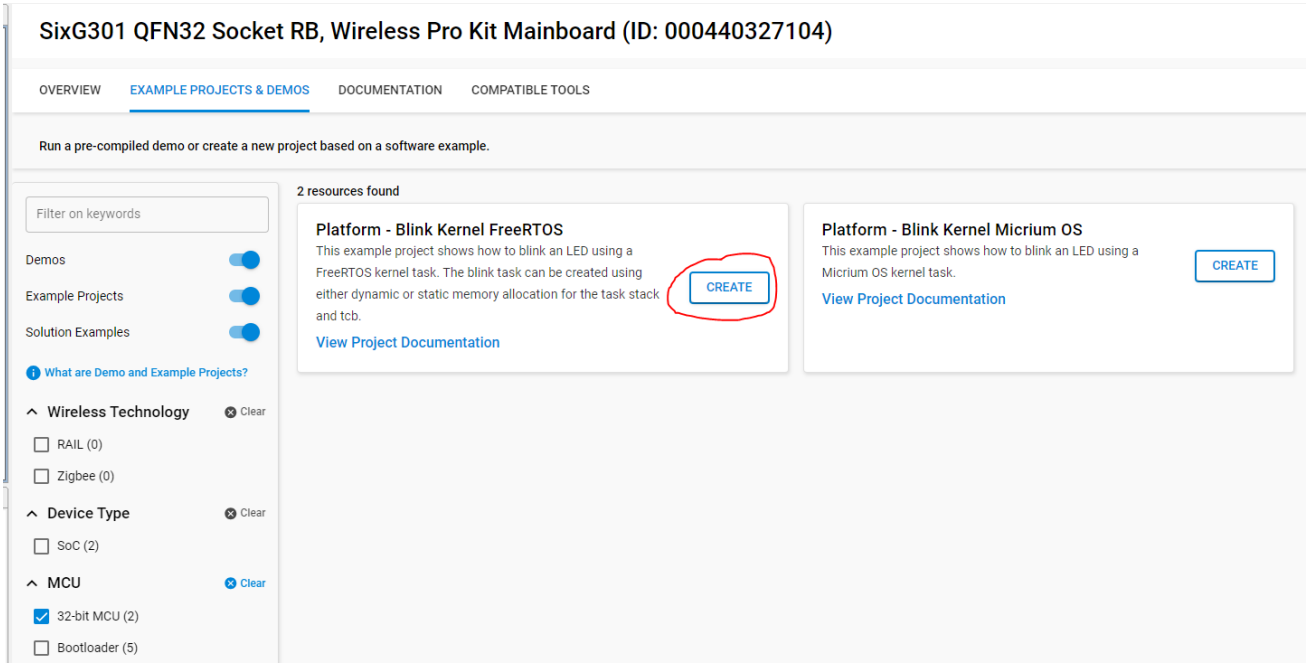
8. Flash the bootloader to the device.

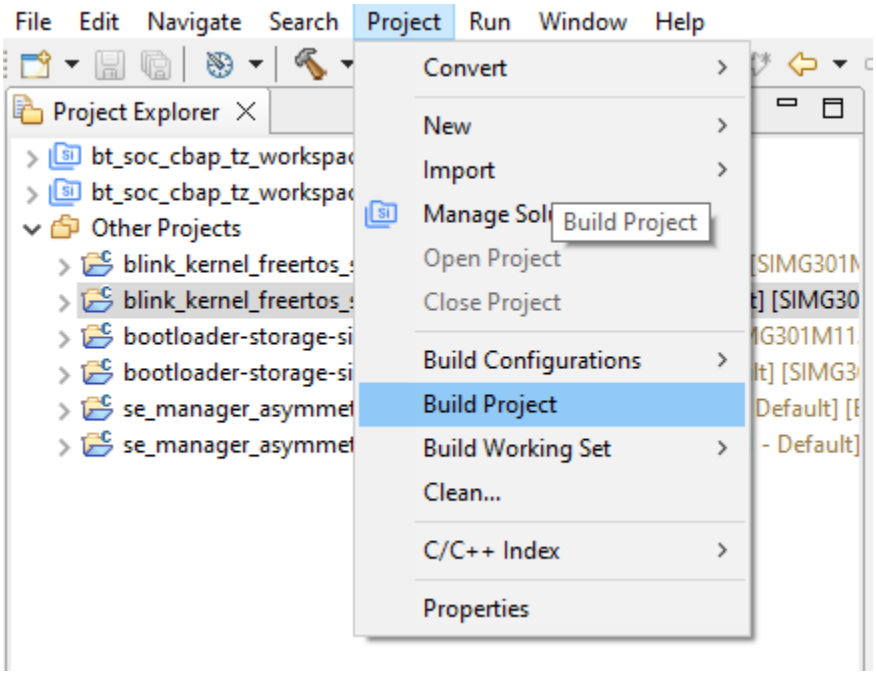
```
commander flash bootloader-storage-single-signed-cert-v1.s37 -d SIMG301M104LIH --serialno
440048205
```

```
WARNING: Failed secure boot detected. Issuing a mass erase before flashing to recover the device...
Parsing file bootloader-storage-single-signed-cert-v1.s37...
Writing 14280 bytes starting at address 0x01000000
Erasing range 0x01000000 - 0x01007FFF (1 sector, 32 KB)
Programming range 0x01000000 - 0x01000FFF (4 KB)
Programming range 0x01001000 - 0x01001FFF (4 KB)
Programming range 0x01002000 - 0x01002FFF (4 KB)
Programming range 0x01003000 - 0x01003FFF (4 KB)
Programming range 0x01004000 - 0x01004FFF (4 KB)
Programming range 0x01005000 - 0x01005FFF (4 KB)
Programming range 0x01006000 - 0x01006FFF (4 KB)
Programming range 0x01007000 - 0x01007FFF (4 KB)
JLinkError: Failed to halt CPU.
Closing region 0 (this consumes one OTP bit, consider --noclose on development/testing devices)
Flashing completed successfully!
DONE
```

9. Reset the device to update the boot status of the device. This step is necessary to prevent Simplicity Commander from performing a masserase in the next section.

10. Build an instance of the sample application as shown below:





11. Sign the resulting executable image with the private BL key.

```
commander convert blink_kernel_freertos_simg301.s37 --secureboot --keyfile sign_key.pem --outfile
signed-blink.s37
```

12. Flash the application to the device.

```
commander flash signed-blink.s37 -device SIMG301M104LIH --serialno 440048205
```

```
Parsing file blink_kernel_freertos_simg301.s37...
Found Application Properties at 0x01018878
Writing Application Properties signature pointer to point to 0x010194f0
Setting signature type in Application Properties: 0x00000001
Image SHA256: 73627f86b624bf0bd52b93e857a7518b0408b71dee2dba639f1fc5c67afff284
R = F553D3DE255786FB2629C93EEC40E03F223C831D9E7F93E2074C4C83294ABA10
S = AC8BD6AF0E680B4590132E3D1317D09E2CEB9EF89434880E6E96369960C724A8
Writing to signed-blink.s37...
Overwriting file: signed-blink.s37...
DONE
```

3.5 Wrapping a Key for Secure Storage

In this example, you'll generate an asymmetric keypair and wrap it securely.

1. Start with any of the available sample applications in the Simplicity SDK.
2. Add the following snippet of code to create storage for the key material and a key descriptor.

```
#define ECC_P256_PRIVKEY_SIZE     (SL_SE_KEY_TYPE_ECC_P256 & SL_SE_KEY_TYPE_ATTRIBUTES_MASK)
#define ECC_P256_PUBKEY_SIZE       2*ECC_P256_PRIVKEY_SIZE
#define OVERHEAD                   (12+16)
#define P256_BUFFER_SIZE (ECC_P256_PRIVKEY_SIZE+ECC_P256_PUBKEY_SIZE+OVERHEAD)

uint8_t key_buffer[P256_BUFFER_SIZE];
sl_se_key_descriptor_t example_ecc_key = {
    .type = SL_SE_KEY_TYPE_ECC_P256,
    .flags = SL_SE_KEY_FLAG_ASYMMETRIC_BUFFER_HAS_PRIVATE_KEY
                | SL_SE_KEY_FLAG_ASYMMETRIC_BUFFER_HAS_PUBLIC_KEY
                | SL_SE_KEY_FLAG_ASYMMMETRIC_SIGNING_ONLY,,
    .storage.method = SL_SE_KEY_STORAGE_EXTERNAL_WRAPPED,
    .storage.location.buffer.pointer = key_buffer,
    .storage.location.buffer.size = sizeof(key_buffer)
};
```

3. Add a call to the SE manager init function.

```
sl_se_init();
```

4. Set up a context for SE manager command and a status variable to check the return value of the SE manager API function then call the generate key function.

```
sl_se_command_context_t ctx;
sl_status_t status;
status = sl_se_generate_key(&ctx,&example_ecc_key);
```

5. Print out the status returned by the API call to confirm that it was successful.

```
printf("result of generating a wrapped key is 0x%2.2X\r\n", status);
```

```
result of generating a wrapped key is 0x00
```

Sign and verify a message with the wrapped key.

1. Create a sample message to be signed and a buffer for the signature

```
char message[] = "message";
uint8_t signature[64];
```

```
result of signing with the wrapped key is 0x00
```

2. Call the API to sign the message using the private part of the key from the previous example and check the return value.

```
status = sl_se_ecc_sign(&ctx,
                        &example_ecc_key,
                        SL_SE_HASH_SHA256,
                        false,    //message is not hashed
                        message,
                        sizeof(message),
                        signature,
                        sizeof(signature));
 printf("result of signing with the wrapped key is 0x%2.2X\r\n", status);
```

3. Call the API to verify the signature of the message using the public part of the key from the previous example.

```
status = sl_se_ecc_verify(&ctx,
                          &example_ecc_key,
                          SL_SE_HASH_SHA256,
                          false,    //message is not hashed
                          message,
                          sizeof(message),
                          signature,
                          sizeof(signature));
  printf("result of verifying with the wrapped key is 0x%2.2X\r\n", status);
```

```
result of verifying with the wrapped key is 0x00
```

3.6 Secure SE Firmware Update

This section describes how to upgrade the SE firmware running on the Series 3 device.

1. Locate the SE firmware folder which is typically C:\Users\<user name> \SimplicityStudio\SDKs\simplicity_sdk\util\se_release\public
2. Sign the SE firmware updater application with your private signing key

```
commander convert --secureboot x301_se_fw_upgrade_app_3v3p1.hex --keyfile rootsign-unsafe-priv-
key.pem --outfile x301_se_fw_upgrade_3v3p1_signed.hex
```

```
Parsing file x301_se_fw_upgrade_app_3v3p1.hex...
Found Application Properties at 0x01028838
Writing Application Properties signature pointer to point to 0x01029334
Setting signature type in Application Properties: 0x00000001
Image SHA256: 605a1afc8196efecc7f87e0b7444ed8268051a8f9d58878a1d64b27c5c5999d2
R = 10267CEBD206C9C1E881C2AD3252EED26416C44B8DCB402C0CBC83679B191026
S = B180DA23EF1E6962B967B27E87332994A57A75E3DB09C1B45C6905F34DBC4A32
Writing to x301_se_fw_upgrade_3v3p1_signed.hex...
DONE
```

3. Flash the signed SE firmware updater application to the device

```
commander flash x301_se_fw_upgrade_3v3p1_signed.hex --device SiMG301M114LIH --serialno
440048205
```

4. Check the version of the SE firmware to confirm that the update was successful

```
commander security status --device SiMG301M114LIH --serialno 440048205
```

```
SE ROM version        : 5.2
SE Firmware version   : 3.3.1
Serial number         : 000000000000000094a081fffe53b457
Debug lock            : Disabled
Device erase          : Enabled
Secure debug unlock   : Disabled
Tamper status         : OK
Secure boot           : Enabled
Boot status           : 0x20 - OK
Command key installed : False
Sign key installed    : True
DONE
```

3.7 Enabling and Working with Secure Debug

The debug lock is an important feature to prevent attackers from using the debug interface to perform illegal operations on the device. The following sections describe how to apply three different locks to the Series 3 debug interface. Only secure debug lock is described here, for other debug lock modes, refer to AN1190.

1. The following command enables the secure debug unlock.

```
commander security lockconfig --secure-debug-unlock enable --device SIMG301M104LIH--serialno
440048205
```

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

2. After enabling the Secure Debug feature, lock the debug interface using the following command.

```
commander security lock --device SIMG301M104LIH--serialno 440048205
```

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

4 Revision History

September, 2025

- Initial release.