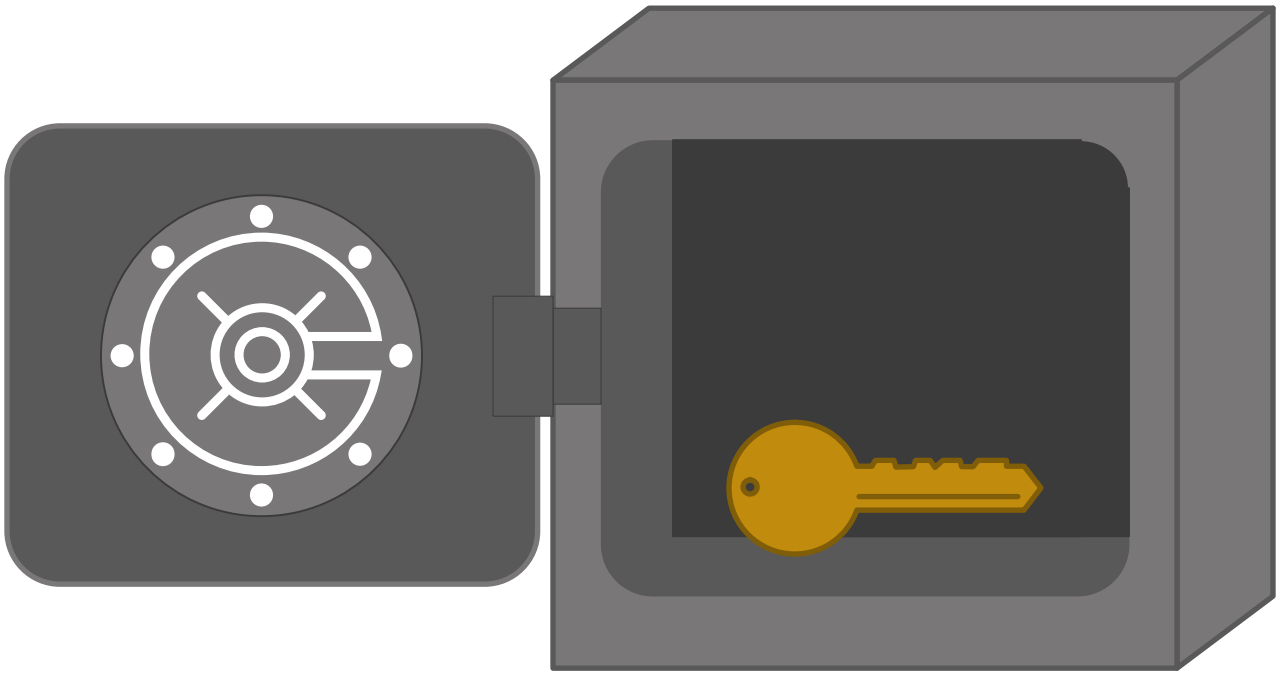
AN1271: Secure Key Storage

Secure Key Storage is a feature in Secure Vault-enabled Series 2 devices that allows for the protection of cryptographic keys by key wrapping. User keys are encrypted by the device's root key for non-volatile storage for later usage. This prevents the need for a key to be stored in plaintext format on the device, preventing attackers from gaining access to the keys through traditional flash-extraction or application attacks, and allowing for a potentially unlimited number of keys to be securely stored in any available storage.

This document describes the operation and usage of this feature, and provides comparisons with other key storage methods.

KEY POINTS

- Keys are encrypted or 'wrapped' with a Secure Vault root key
- Secure Vault root key is not stored on the device, instead it is generated on each reset
- Wrapped keys are confidential to the Secure Vault, and can be stored in NVM safely
- Wrapped keys can be imported into Secure Vault for usage at a later time



1. EFR32 Series 2 Device Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options to help developers build secure devices, secure application software, and secure paths of communication to manage those devices. Silicon Labs' security offerings were significantly enhanced by the introduction of the EFR32 Series 2 products that included a Secure Vault. The Secure Vault is a tamper-resistant component used to securely store sensitive data, keys and to execute cryptographic functions and secure services.

The Secure Vault is the foundation of two core security functions:

- Secure Boot: Process where the initial boot phase is executed from an immutable memory (such as ROM) and where code is authenticated before being authorized to be executed.
- Secure Debug access control: The ability to lock access to the debug ports for operational security, and to securely unlock them when access is required by an authorized entity.

Some EFR32 Series 2 products offer additional security options through Secure Vault. Secure Vault is a dedicated security CPU that isolates cryptographic functions and data from the host processor core. Devices with Secure Vault offer the following security features:

- Secure Key Storage: Protects cryptographic keys by "wrapping" or encrypting the keys using a root key known only to the Secure Vault.
- Anti-Tamper protection: A configurable module to protect the device against tamper attacks.
- Device authentication: Functionality that uses a secure device identity certificate along with digital signatures to verify the source or target of device communications.

A Secure Element Manager and other tools allow users to configure and control their devices both in house during testing and manufacturing, and after the device is in the field.

1.1 User Assistance

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

Document	Summary	Applicability
AN1190: Series 2 Secure Debug	How to lock and unlock EFR32 Series 2 debug access, including background information about the Secure Element	EFR32 Series 2
AN1218: Series 2 Secure Boot with RTSL	Describes the secure boot process on EFR32 Series 2 devices using Secure Element	EFR32 Series 2
AN1247: Anti-Tamper Protection Configuration and Use	How to program, provision, and configure the anti-tamper module	EFR32 Series 2 with Secure Vault
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	EFR32 Series 2 with Secure Vault
AN1271: Secure Key Storage (This document)	**How to securely "wrap" keys so they can be stored in non-volatile storage.**	**EFR32 Series 2 with Secure Vault**
AN1222: Production Programming of Series 2 Devices	How to program, provision, and configure security information using Secure Element during device production	EFR32 Series 2

1.2 Key Reference

Public/Private keypairs along with other keys are used throughout Silicon Labs security implementations. Because terminology can sometimes be confusing, the following table lists the key names, their applicability, and the documentation where they are used.

Table 1.1.

Key Name	SE Manager ID	Customer Programmed	Purpose	Used in
Public Sign key (Sign Key Public)	SL_SE_KEY_SLOT_APPLICATION_SECURE_BOOT_KEY	Yes	Secure Boot binary authentication and/or OTA upgrade payload authentication	AN1218 (primary), AN1222
Public Command key (Command Key Public)	SL_SE_KEY_SLOT_APPLICATION_SECURE_DEBUG_KEY	Yes	Secure Debug Unlock or Disable Tamper command authentication	AN1190 (primary), AN1222, AN1247
OTA Decryption key (GBL Decryption key) aka AES-128 Key	SL_SE_KEY_SLOT_APPLICATION_AES_128_KEY	Yes	Decrypting GBL payloads used for firmware upgrades	AN1222 (primary), UG266
Attestation key aka Private Device Key	SL_SE_KEY_SLOT_APPLICATION_ATTESTATION_KEY	No	Device authentication for secure identity	AN1268

2. Device Compatibility

This application note supports Series 2 device families with the Secure Vault feature. Some functionality is different depending on the device.

Wireless SoC Series 2 devices with Secure Vault consists of:

- EFR32BG21B
- EFR32MG21B

3. Secure Element Manager

The Secure Element Manager provides thread-safe APIs for the Secure Vault's mailbox interface. The Secure Element Manager APIs related to Secure Key Storage operations are listed in the table below. Documentation for Secure Element Manager APIs can be found at https://docs.silabs.com/gecko-platform/latest/service/api/group-sl-se-manager.

Table 3.1. Secure Element Manager API for Secure Key Storage

Secure Element Manager API	Usage
`sl_se_generate_key`	Generate a new key into a Secure Vault volatile key storage slot.
`sl_se_transfer_key`	Transfer a wrapped key. Used to export a wrapped key from an internal storage slot or an external plaintext key. Also used to import a wrapped key into the Secure Vault.
`sl_se_export_key`	Export a plaintext key if allowed. This will fail for a key that has been flagged as `SL_SE_KEY_FLAG_NON_EXPORTABLE`.

4. Introduction

Secure Vault is a dedicated security CPU that isolates cryptographic functions and data from the host Cortex-M33 core. It is used to accelerate cryptographic operations as well as to provide a method to securely store keys. This application note will cover the Secure Key Storage feature of the Secure Vault.

The Secure Vault contains one-time programmable memory (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 OTA Decryption key, used for Over-the-Air updates

These keys are one-time programmable, and, after programming, are persistent on the device through a power-on reset.

The Secure Vault also contains four volatile storage slots for any 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 outside of the Secure Vault in non-volatile storage. After a device reset, the key can be loaded into the Secure Vault volatile key storage for usage by index, or used in-place (passed to the Secure Vault on every requested operation). Without any secure key storage mechanism, this necessitates the storage of the user key in plaintext format in non-volatile storage. This opens the keys 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 export a key from the Secure Vault in 'wrapped' format. In this format, the key is encrypted by a device-unique root key, only available to the Secure Vault. 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 into the Secure Vault for usage without ever exposing the plaintext key to the application, which also prevents application-level attacks from exposing the key.

5. Key Storage Method Comparisons

The following section demonstrates three methods for key storage: plaintext, ARM® TrustZone®, and Secure Key Storage.

Note: In the following examples, AES key usage is demonstrated. However, any other key types supported by the device can also be used with Secure Vault key storage.

5.1 Key Generation and Usage

In Series 2 devices, cryptographic functions are performed by the Secure Vault. In order to perform these functions, the Secure Vault must have access to any user keys needed. Keys can be generated and be used by the Secure Vault in multiple ways:

1. External storage, in-place usage:
   a. A user generates a plaintext key and stores it in device memory.
   b. The user provides a key descriptor to the Secure Vault that points to this key for a specific cryptographic operation.
   c. The Secure Vault performs the cryptographic operation using this key, but does not store it in any Secure Vault volatile storage slot.
2. External storage with Secure Vault import:
   a. A user generates a plaintext key and stores it in device memory.
   b. The user provides a key descriptor to the Secure Vault that points to this key, as well as a slot number to store the key.
   c. The Secure Vault imports this key into a volatile key storage slot.
   d. The user requests that the Secure Vault performs a cryptographic function by providing the index of the storage slot.
3. Internal Secure Vault key generation:
   a. The user commands the Secure Vault to generate a new key within one of the Secure Vault's volatile key slots.
   b. The user requests that the Secure Vault performs a cryptographic function by providing the index of the storage slot.

In each case, in order to provide persistent storage for the key, the key must be stored in non-volatile memory.

5.2 Plaintext Key Storage

The simplest manner to store a key is to save it in plaintext form. The steps to store and use a key stored in plaintext form are as follows:

1. A user key is generated and imported into device memory. For persistent storage, this must be non-volatile storage, such as device flash.
2. After a device reset, the Secure Vault volatile key storage will be empty, so the plaintext key must be loaded into a slot for usage. Alternatively, the key could be used in-place from non-volatile storage on a per-operation basis.

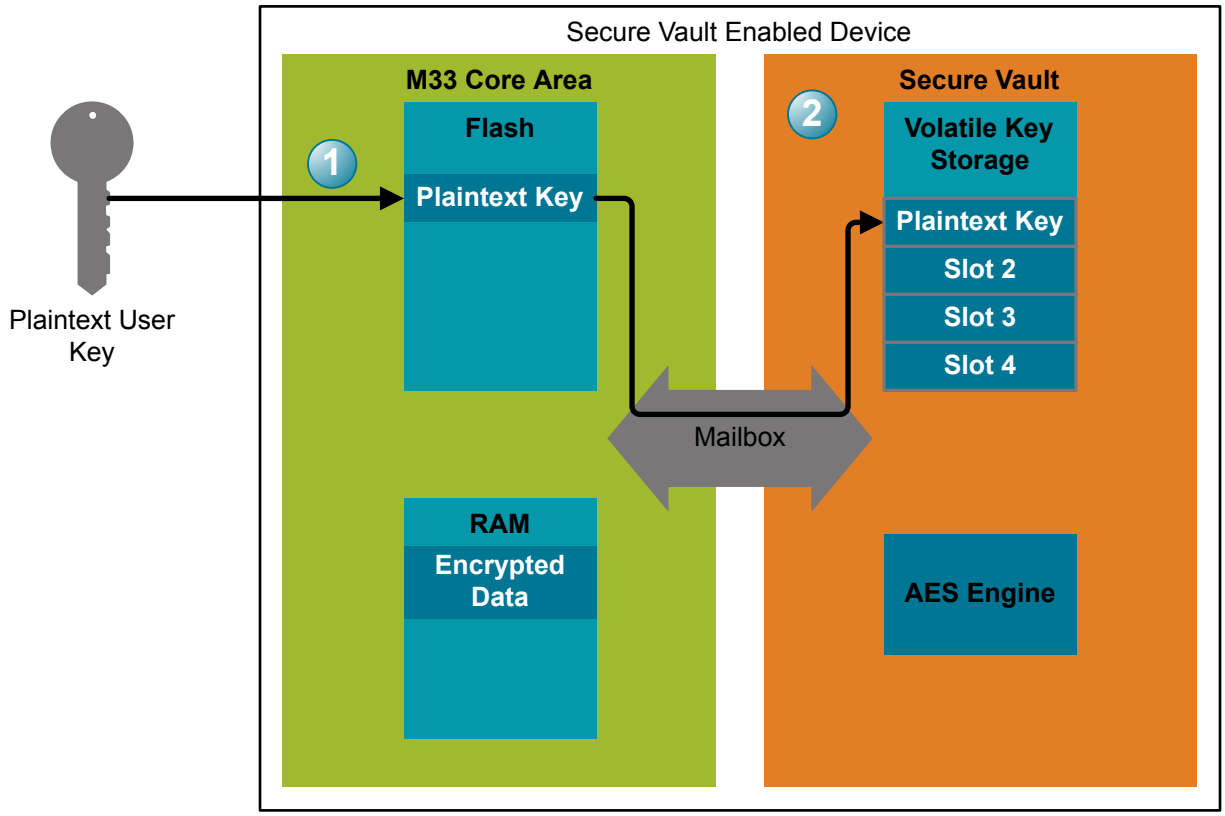


Figure 5.1. Plaintext Key Storage

In order to use the key for a cryptographic operation, the following procedure is used.

1. The user passes data to be processed (in this specific example, AES encrypted data) to the Secure Vault.
2. The user requests a that cryptographic operation be performed on this data using one of the keys stored in the Secure Vault volatile key storage slots. Alternatively, the key can be passed to the Secure Vault directly for a singular cryptographic operation.
3. The Secure Vault performs the cryptographic operation.
4. The output of the cryptographic operation is passed back to the user for processing.

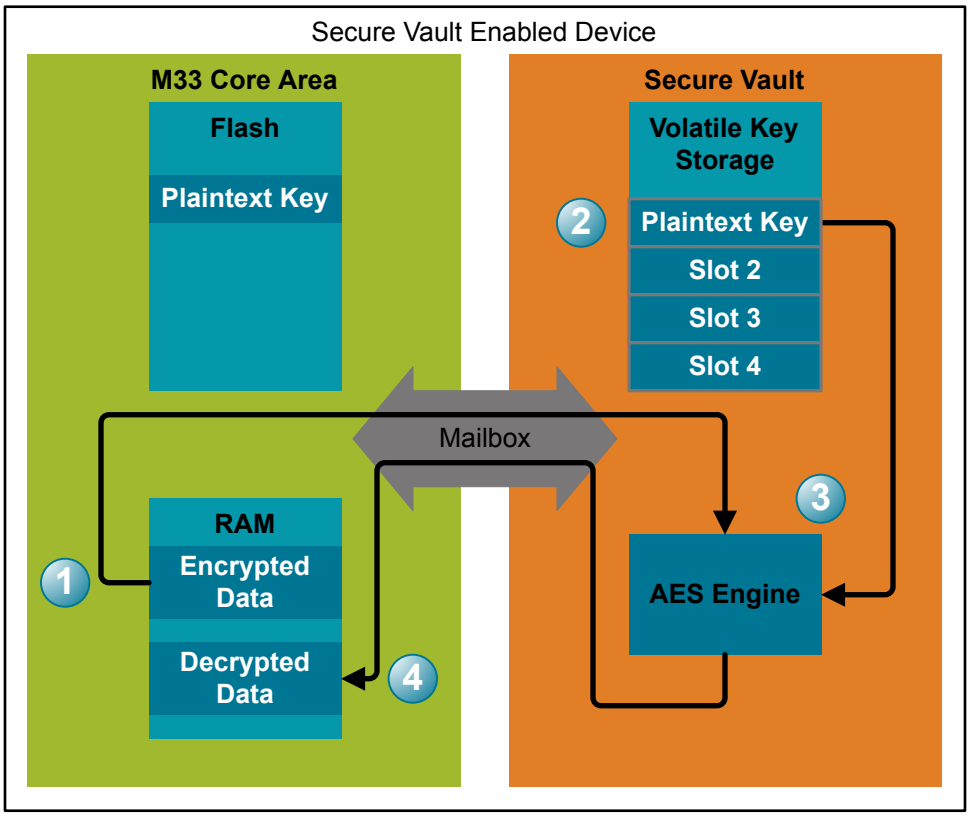


Figure 5.2. Plaintext Key Usage

This method exposes the keys to two major vulnerabilities:

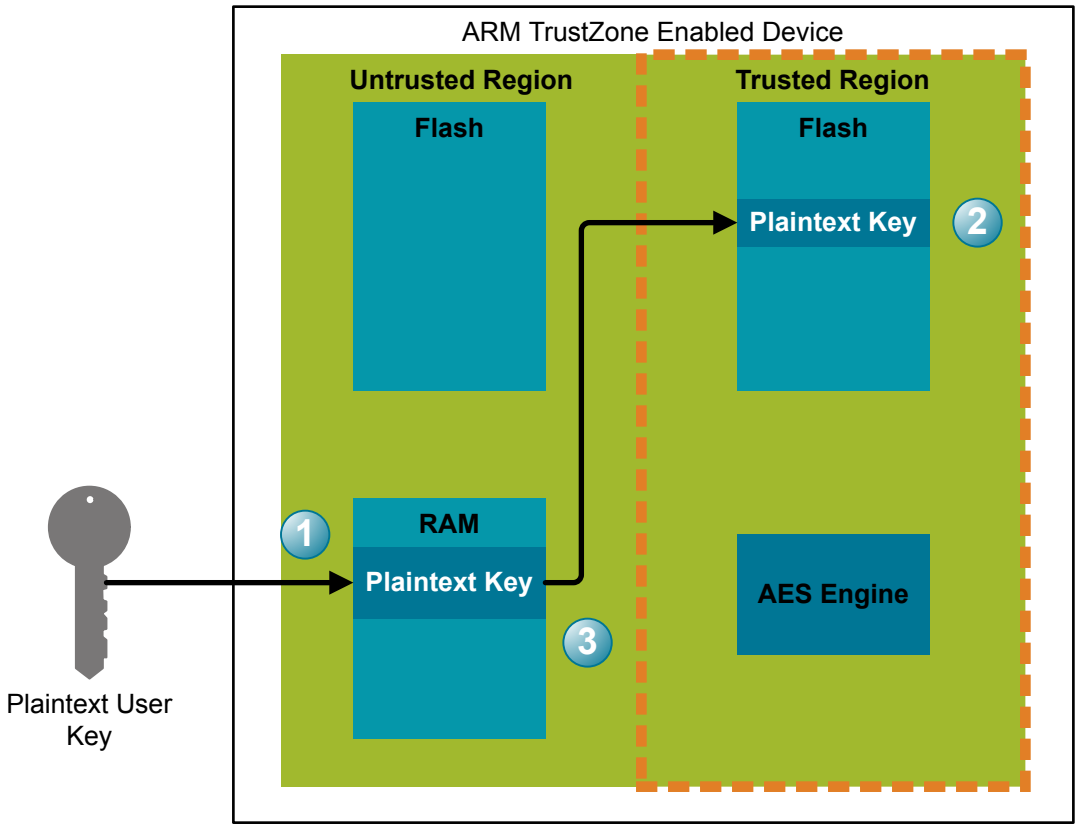
1. Access to device storage gives access to the keys. In this case, an attack that gains access to the flash contents will expose the user key.
2. Since the application has access to the keys, compromising the application or device privileges can compromise the keys. Such an attack might not directly access device memory, but take control of the application in a way that causes the application to expose the key to an attacker.

5.3 ARM® TrustZone® Key Storage

In some ARM devices, key management is handled by a feature called TrustZone. TrustZone divides the device memory map into two regions: a trusted region and a non-trusted region. User code is executed from the non-trusted region, which does not have access to any part of the trusted region. The trusted region is used to store user keys securely and to control other secure operations. Using TrustZone for key storage requires the following steps:

1. A plaintext user key is created and imported into untrusted device memory.
2. The key is imported into the trusted region from the untrusted region. For persistent storage, the key must be stored in device non-volatile memory, such as flash.
3. The plaintext key in the untrusted region can now be deleted. From here, the untrusted region no longer has direct access to the plaintext key.

Figure 5.3. TrustZone Key Import



Once a key has been loaded into the trusted zone, it can be used for cryptographic operations through the following procedure:

1. The user passes data to be processed (in this specific example, AES encrypted data) to the trusted zone.
2. The user requests that a cryptographic operation be performed on this data using one of the keys stored in the trusted region.
3. The trusted region performs the cryptographic operation.
4. The output of the cryptographic operation is passed back to the untrusted region for processing.

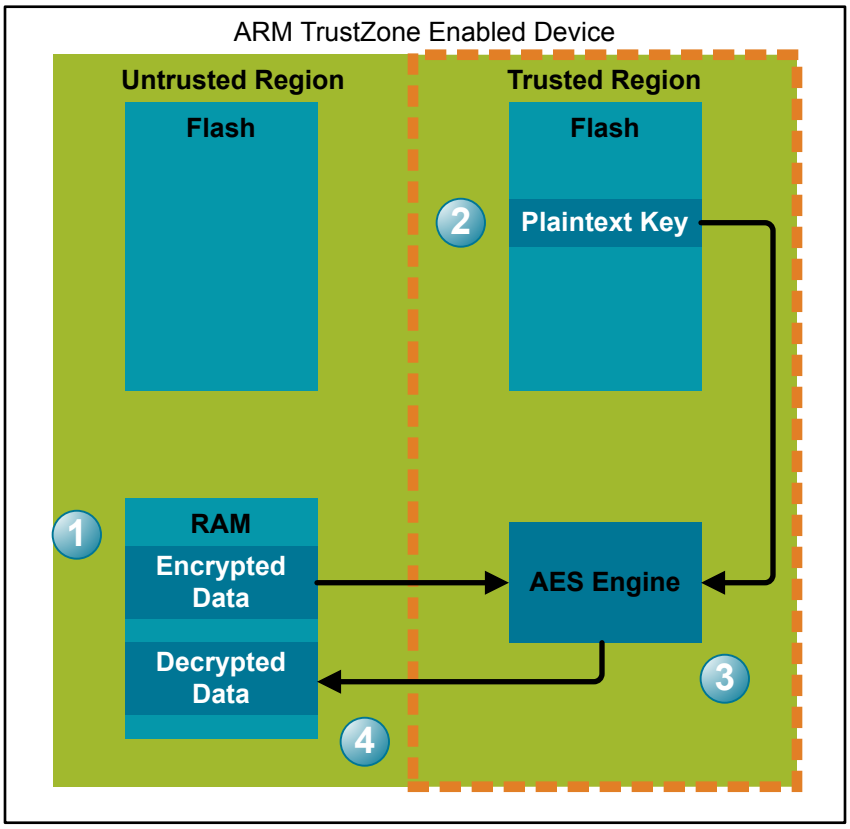


Figure 5.4. TrustZone Key Usage

Once the key is loaded into the trusted region, the user application no longer has direct access to the key. This eliminates the application-level vulnerability mentioned previously. However, the plaintext version of the key is still stored in device memory, meaning that this method is still vulnerable to a storage extraction attack on the device.

TrustZone also has another limitation: since the trusted area is some portion of the device memory, the number of keys that you can securely store is limited to the space available within the trusted region.

5.4 Secure Key Storage

With Secure Key Storage, the user key, using the Secure Vault, can be exported in an encrypted, or 'wrapped' form. Only the Secure Vault has access to the key to decrypt, or 'unwrap', the wrapped key. This Secure Vault root key is not stored on the device during power-down, but rather regenerated after each reset. This allows a user to securely store a key in non-volatile memory, limiting the number of keys that can be stored only by the amount of storage the user has available.

To wrap an externally-generated key:

1. After power-on, the device's unique root key is generated with output from the PUF (Physically Unclonable Function).
2. A user key is generated and imported into device memory. In this example, the key is imported into RAM for easy deletion, and the added security that, if device power is removed, the key will be lost.
3. The user key is passed to the Secure Vault, where it is encrypted with the Secure Vault's root key.
4. The wrapped key is passed back to the user application for storage in non-volatile memory (in this case, device flash).
5. The plaintext key can now be deleted from the device. From this point forward, only the Secure Vault will have access to the plaintext key.

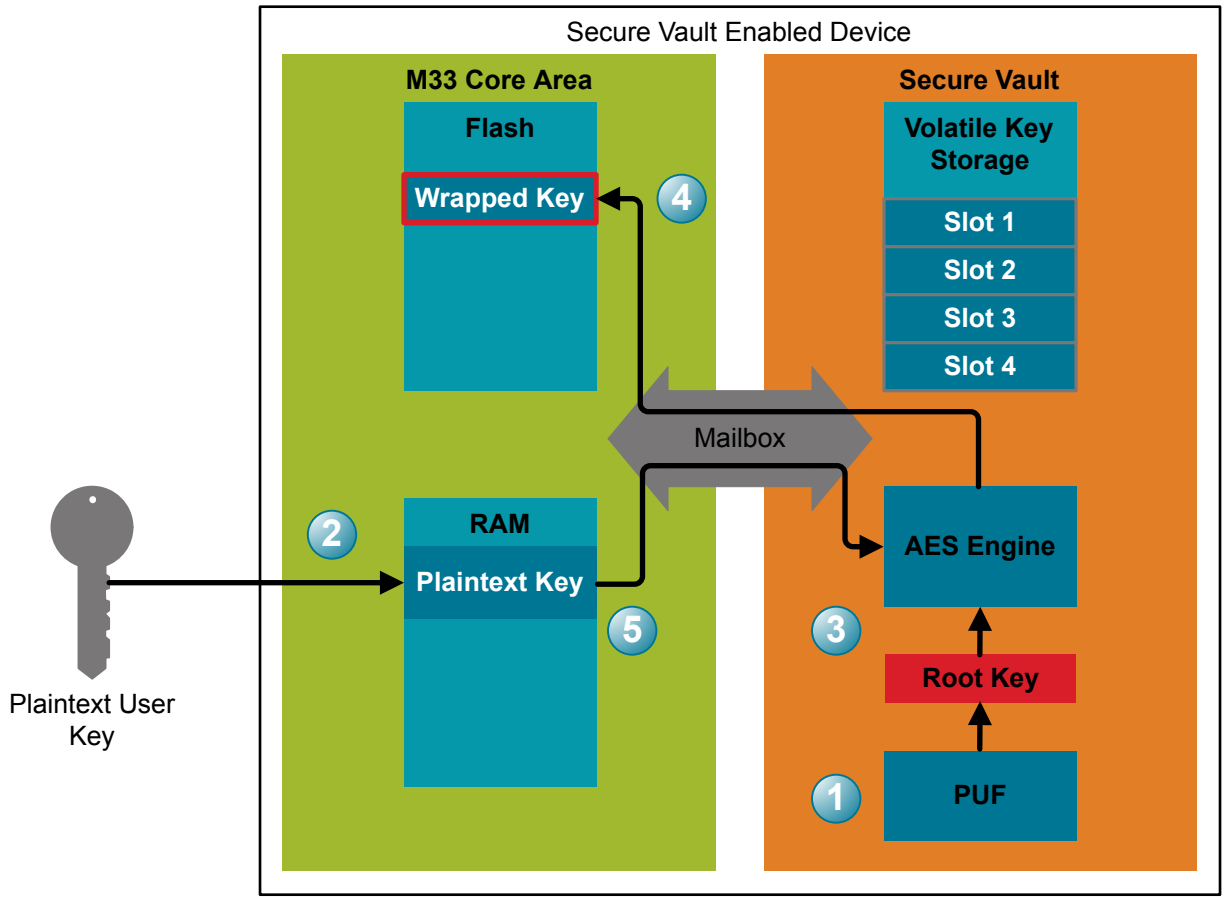


Figure 5.5. External Key Import, Wrapping, and Storage

Instead of importing an external key, the Secure Vault can generate a new key directly into one of its volatile key storage slots. This key can then be exported in wrapped form for secure persistent storage.

1. The user requests that the Secure Vault generates a new key into one of its storage slots using the true random number generator (TRNG).
2. The key is encrypted with the Secure Vault's root key.
3. The wrapped key is passed back to the user application for non-volatile storage (flash, in this case).

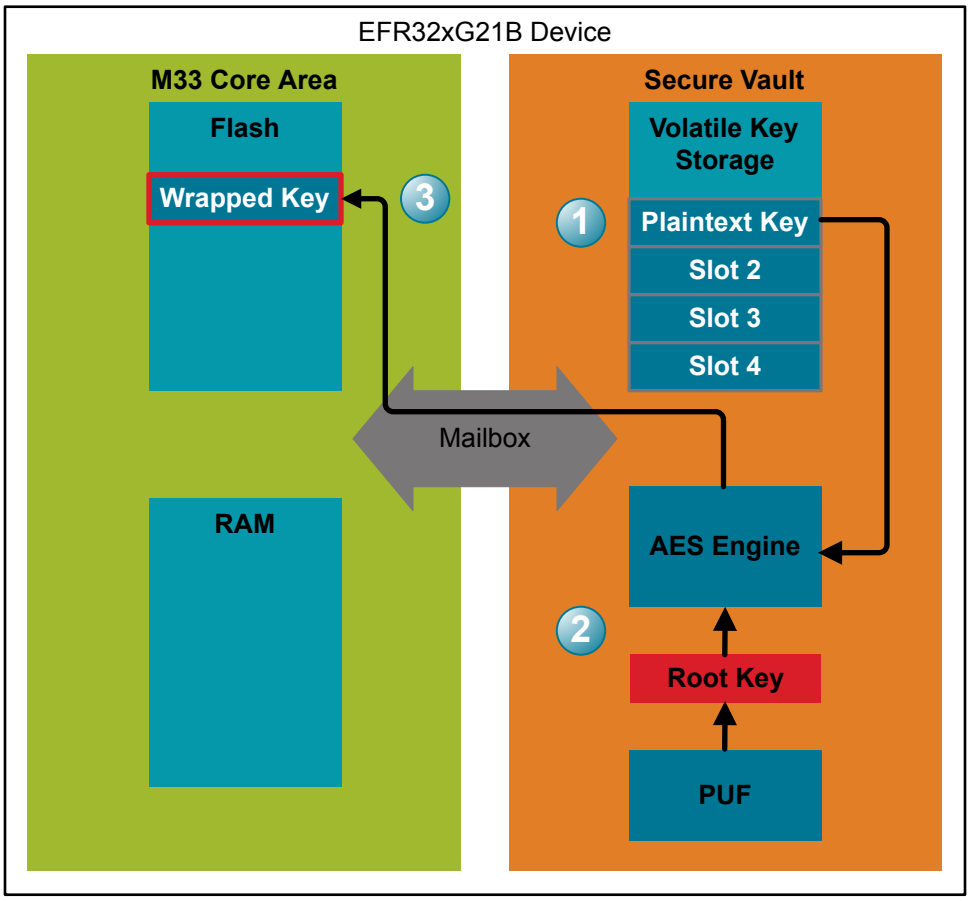


Figure 5.6. Internally Generated Key Wrapping and Storage

In order to import a wrapped key into the Secure Vault for usage:

1. The wrapped key is passed to the Secure Vault.
2. The wrapped key is decrypted ("unwrapped") with the Secure Vault's root key.
3. The plaintext key is stored in a volatile key storage slot.

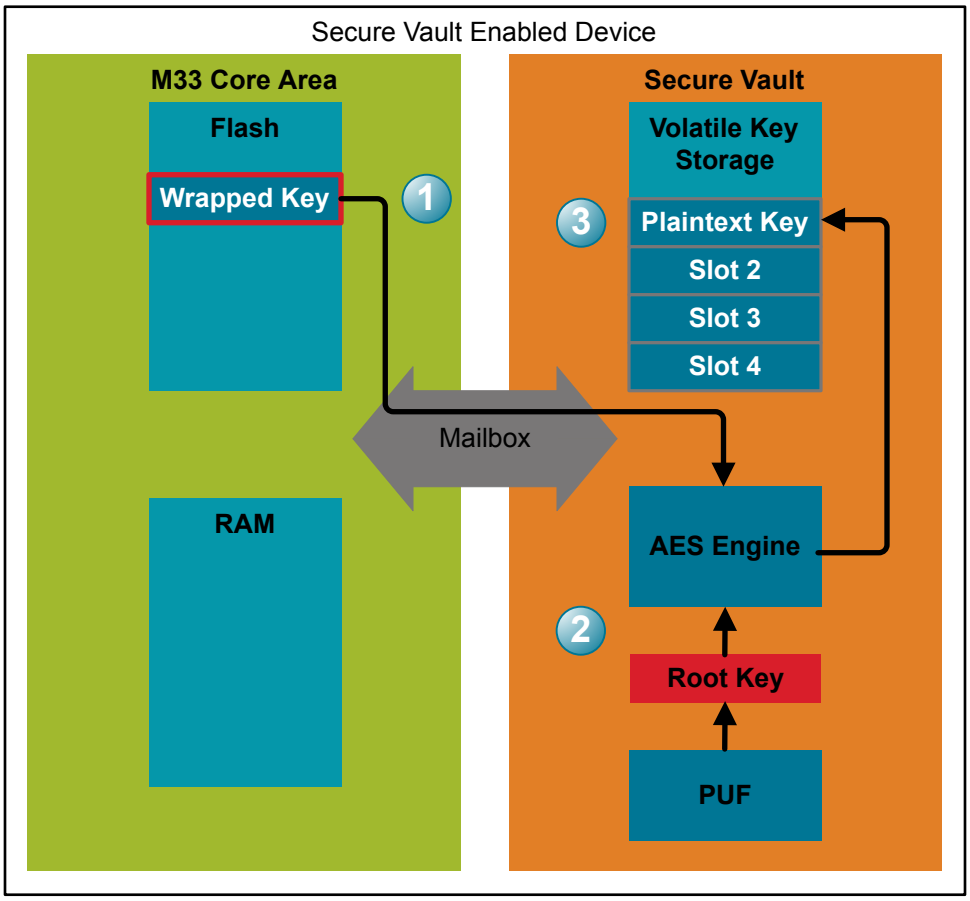


Figure 5.7. Wrapped Key Import

In order to use the key for a cryptographic operation, the same steps are followed as when using a plaintext key that has been imported into the Secure Vault:

1. The user passes data to be processed (in this specific example, AES encrypted data) to the Secure Vault.
2. The user requests that a cryptographic operation be performed on this data using one of the keys stored in the Secure Vault volatile key storage slots. Alternatively, the wrapped key can be passed to the Secure Vault directly for a singular cryptographic operation. In this case, the key will be unwrapped before being used, but will not be stored for future operations.
3. The Secure Vault performs the cryptographic operation.
4. The output of the cryptographic operation is passed back to the user for processing.

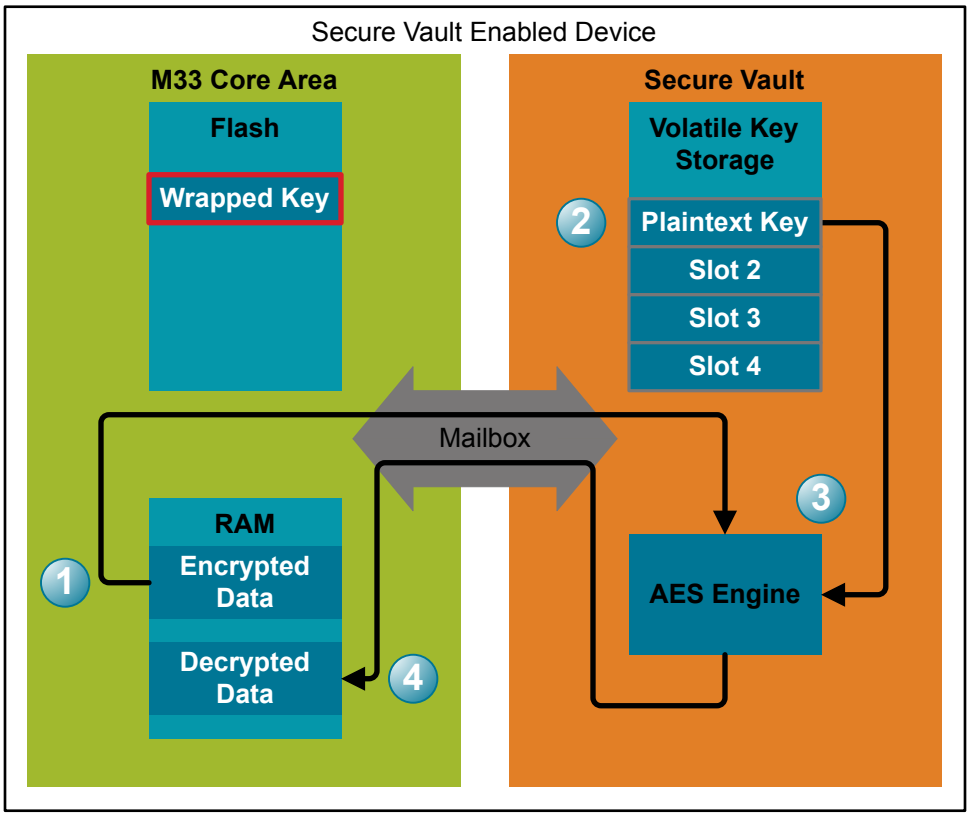


Figure 5.8. Wrapped Key Usage

5.5 Secure Key Storage Advantages

Secure Key Storage confers the following benefits over other key storage methods:

1. Access to device memory does not expose user keys.
2. Compromising the user application does not expose user keys, since the user application itself does not have access to the plaintext keys.
3. The number of user keys that can be securely stored is only limited by the amount of storage available to the user, including external storage.

6. Operation Details

6.1 Root Key Generation

Secure Key Storage depends on the Secure Vault to encrypt / decrypt (wrap / unwrap) user keys with its own symmetric root key. The symmetric key used for this wrapping and unwrapping must be highly secure as it can expose all other key material in the system. The Secure Vault Key Management system uses a Physically Unclonable Function (PUF) to generate a persistent device-unique seed key on power up to dynamically generate this critical wrapping/unwrapping key. The key is only visible to the AES encryption engine and it is not retained when the device loses power.

6.2 Exporting a Wrapped Key

Using Secure Key Storage, a user key can be set to non-exportable in its key descriptor. This prevents the Secure Vault from exporting the plaintext key when it is requested. However, a 'wrapped' version of the key can still be exported to the user application. On export, the key is first encrypted by the Secure Vault's root key. The Secure Vault also tags the key with information to identify the wrapped key. Since only the Secure Vault has access to use the root key, the plaintext key is non-accessible, even to the user application.

Note: Wrapped keys are slightly larger than the equivalent plaintext key, as some additional metadata is required to identify the wrapped key to the Secure Vault.

6.3 Wrapped Key Storage and Usage

Once a key has been wrapped and exported, it can be safely stored anywhere - device flash, RAM, external storage, etc. The number of keys that can be securely stored is only limited by the available storage space. A wrapped key can later be imported into a Secure Vault volatile storage slot for usage, or used in-place. Once the key is wrapped and stored, the plaintext key available to the application can be deleted. From here, only the Secure Vault will have the ability to unwrap and use the key.

With access to the wrapped key, the Secure Vault can use this key in one of two ways:

1. A user can request that a cryptographic operation be performed using the key stored in memory. In this case, the Secure Vault will import the key, unwrap it, and then perform the cryptographic operation. The key will not be stored within the Secure Vault.
2. A user can import the wrapped key into a Secure Vault volatile storage slot. In this case, the key is unwrapped by the Secure Vault and stored in plaintext in a volatile slot. The user can then later request that a cryptographic function be performed by the Secure Vault by referencing the volatile slot index. This provides a performance increase over using wrapped keys in place, as the Secure Vault does not need to import and unwrap the key on each requested operation.

6.4 Password Protection

When defining a key descriptor for a new key, or when importing an existing key into Secure Vault, the user can choose to require a password to allow use of the key. The password field in the key descriptor structure is eight bytes in length. If unspecified, the key will use the default password of all zeros.

After importing a key with a password, failing to provide the correct password when performing a cryptographic operation will result in Secure Vault returning an invalid credentials error, and no operation will be performed.

7. Examples

7.1 Simple Key Generation and Storage

This example demonstrates the generation, storage, and use of a wrapped AES-256 key using Secure Element Manager.

Key Generation

A new AES-256 key can be generated into the Secure Vault volatile key storage slot 0 by the following code. It is important to set the key flags to include `SL_SE_KEY_FLAG_NON_EXPORTABLE` to disallow the Secure Vault from exporting the plaintext key.

```
sl_status_t status;
sl_se_command_context_t ctx;

sl_se_key_descriptor_t key = {
    .type = SL_SE_KEY_TYPE_AES_256,
    .flags = SL_SE_KEY_FLAG_NON_EXPORTABLE,
    .storage.method = SL_SE_KEY_STORAGE_INTERNAL_VOLATILE,
    .storage.location.slot = SL_SE_KEY_SLOT_VOLATILE_0 };

status = sl_se_generate_key(&ctx, &key);
```

Key Usage

This key can now be used for cryptographic operations. For example:

```
unsigned char msg[] = "Hello world.";
unsigned char msgEncrypted[16];

status = sl_se_aes_crypt_ecb(&ctx,
                        &key,
                        SL_SE_ENCRYPT,
                        msg,
                        msgEncrypted);
```

Note, this example only encrypts one AES block. For AES-256, the block size is 16 bytes.

Wrapped Key Export

To save the wrapped key to flash, first export it to a RAM buffer.

```
unsigned char aesKeyWrappedBuf[32 + 28];

sl_se_key_descriptor_t keyWrapped = {
      .type = SL_SE_KEY_TYPE_AES_256,
      .flags = SL_SE_KEY_FLAG_NON_EXPORTABLE,
      .storage.method = SL_SE_KEY_STORAGE_EXTERNAL_WRAPPED,
      .storage.location.buffer.pointer = aesKeyWrappedBuf,
      .storage.location.buffer.size = sizeof(aesKeyWrappedBuf) };

status = sl_se_transfer_key(&ctx, &key, &keyWrapped);
```

Note: A wrapped key will always add an additional 28 bytes of overhead to the key size. In this example, the AES-256 key, unwrapped, is 32 bytes. Wrapped, it is 32 + 28 bytes.

Writing Wrapped Key to Flash

Once a key has been exported to a RAM buffer, the MSC can be used to write it to a flash location.

```
#define KEY_STORAGE_ADDR (0xF0000)
MSC_WriteWord((uint32_t *)KEY_STORAGE_ADDR, aesKeyWrappedBuf, sizeof(aesKeyWrappedBuf));
```

In this example, the key is written to address 0xF0000.

Importing Wrapped Key

After a device reset, or if the key has been deleted from its volatile slot, the key can be imported back into slot 0 by the following:

```
sl_status_t status;
sl_se_command_context_t ctx;

sl_se_key_descriptor_t key = {
    .type = SL_SE_KEY_TYPE_AES_256,
    .flags = SL_SE_KEY_FLAG_NON_EXPORTABLE,
    .storage.method = SL_SE_KEY_STORAGE_INTERNAL_VOLATILE,
    .storage.location.slot = SL_SE_KEY_SLOT_VOLATILE_0 };

sl_se_key_descriptor_t keyWrapped = {
        .type = SL_SE_KEY_TYPE_AES_256,
        .flags = SL_SE_KEY_FLAG_NON_EXPORTABLE,
        .storage.method = SL_SE_KEY_STORAGE_EXTERNAL_WRAPPED,
        .storage.location.buffer.pointer = (unsigned char*)KEY_STORAGE_ADDR,
        .storage.location.buffer.size = 32+16 };

status = sl_se_transfer_key(&ctx, &keyWrapped, &key);
```

7.2 Github Examples

Additional Secure Element Manager examples can be found in the Silicon Labs Github repository under the series2/semgr/semgr_secure_key_storage directory. The following examples are available:

generate_AES_export_wrapped

This example generates a new AES-256 key into a Secure Vault volatile storage slot. The key is used to encrypt a message, which is then stored in flash. The key is then exported in wrapped form and stored into flash. On device reset, if a wrapped key is stored in flash, it is loaded into the Secure Vault, and used to decrypt the message. The key in flash is then erased.

generate_AES_export_wrapped_with_password

The previous example is duplicated with the addition of an 8-byte password used to protect access to the key.

import_rand_AES_export_wrapped

The first example is duplicated. Instead of using Secure Vault to generate a key within a storage slot, a random key is generated using the TRNG in a RAM buffer. This buffer is imported as an AES key into a volatile storage slot, then deleted from RAM. The key is then exported in wrapped format.

ecdh_export_wrapped

An ECDH exchange is performed to generate a 32-byte shared secret. The shared secret is used as an AES-256 key to encrypt and decrypt a message. The AES key is then exported in wrapped format. At no point is a plaintext version of the generated shared secret available to the user.

import_ECC_P256_pem_export_wrapped

This example demonstrates importing a PEM-encoded key into the device. PEM is a common key file format that allows encoding a key into ASCII characters. An example PEM encoded private EC key is shown below.

```
-----BEGIN EC PRIVATE KEY-----
MHcCAQEEIAG+QwyHX1dmKtkBGKiOjEfDvfZH+kd4pTtLOmoa1tEYoAoGCCqGSM49
AwEHoUQDQgAErpxARso4TaKuxCObCG3G1eUNxLVsdkzHB9UJi1BK0xQobLq0Nj3H
DKh0vb/PBoaWw1kTx+P1HcnGUU4GTOpniw==
-----END EC PRIVATE KEY-----
```

In this example, the user generates a key pair using openssl (run openssl_genkeys.sh in the base example directory). Example key pairs have also been provided in the base example directory as private.pem/der and public.pem/der. The EC private key is imported in PEM format over UART. mbedtls is used to convert the PEM-encoded key into binary format, which is then imported into a Secure Vault volatile slot. The plaintext keys are then deleted from RAM, and the wrapped key is exported and stored in flash.

This example uses OpenSSL to generate PEM encoded keys. The Windows version of OpenSSL can be downloaded from here - https://slproweb.com/products/Win32OpenSSL.html

8. Revision History

Revision 0.1

September 2020

- Initial Revision.

Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

SILICON LABS

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

http://www.silabs.com