This application note explains the process for ordering custom parts through the Custom Part Manufacturing Service (CPMS). Instructions for customizing device identity security certificates and wrapping custom keys are also included.

**What is CPMS?**

Custom Part Manufacturing Service (CPMS) allows you to customize Silicon Labs hardware – wireless SoCs, modules, MCUs – at the factory. The CPMS self-service web portal guides you through the customization process and its various customizable features and settings. You can place orders for customized test and production units to our factories securely via the CPMS portal.

Unlike traditional flash programming, CPMS is a secure provisioning service that enables you to customize your chips with several highly advanced features such as secure boot, secure debug, encrypted OTA, public, private and secret keys, secure identity certificates, and more.

The custom features, identities and certificates are injected on the hardware securely, quickly, and cost-efficiently at the world’s safest place, the Silicon Labs factories.

**Why CPMS?**

Securing an IoT device is a highly complicated and costly process - you must generate public and private keys for secure boot and secure debug, sign code with a private key, store all the private keys in a Hardware Security Module (HSM), place the public keys for secure boot and secure debug in one-time-programmable (OTP) memory, flip OTP bits for secure boot and secure debug, and flash the encrypted code and identity certificates within the hardware. CPMS streamlines the programming part of this process for you. Even the most advanced security features, certificates, and identities can be programmed in a secure, fast, and cost-efficient way at the Silicon Labs factories.

**KEY POINTS**

This application note explains how to:
- Start a new custom part
- Customize the following four fields in the device certificate:
  - Common name
  - Organization
  - Country
  - Organizational unit
- Import custom wrapped keys
1. Custom Certificates

CPMS allows you to customize the device identity certificate chain. The certificates use the X.509 format, and must conform to RFC-3280. At this time, CPMS supports customization of four fields in the device certificate:

1. Common name: User-defined, 30-character name that will terminate with the 64-bit EUI of the device (example is "EUI:xxxxxxxxxxxxxxx" and will terminate with " S:SE0 ID:MCU" or " S:FL0 ID:MCU" depending on if the device is a Secure Vault High device or not.)
2. Organization: User-defined, 64-character company name
3. Country: Must be a legitimate country code letter pair (e.g., US)
4. Organizational Unit: User-defined field of up to 64 characters

If there are other certificate customizations you would like to implement, specify them in the "Special Instructions" section in the CPMS.

Custom Identity

Custom Identity allows customers to extend the default Silicon Labs certificate identity cert chain to provide your own. This is an advanced feature which requires additional charges. Please contact a Silicon Labs sales representative for details.

Read more about secure identity

Scope of Customization

- Device certificate only
- The certificate chain

Special Instructions

Tell us how you would like to customize the identity of this part. (2000/2000 remaining)
2. Key Wrapping

Secure Vault High devices support Key Wrapping, which is a feature where keys are encrypted using a Physically Unclonable Function (PUF) key. A PUF key is secret, random, and unique to each individual device. PUF keys do not live in flash and are not vulnerable to flash extraction attacks.

CPMS allows customers to provide their own keys, which will be wrapped by the secure element and stored on the device. This means that the firmware image does not need to contain the key at any point in production.

To use this feature, you need to provide CPMS with four fields:

1. Key Auth - an 8-byte password that must be provided by software whenever the key is used. This password can be disabled by setting the Key Auth to 0x0000000000000000.
2. Key Value - the value of the key to be wrapped (max 200 bytes).
3. Key Metadata - 4 bytes of key metadata, including information such as the type of key, allowed uses, length, etc. More information on how to generate this value for an existing key can be found in section 3.2 Importing Custom Wrapped Keys.
4. Key Address - the address in user flash to which the key should be programmed.
3. CPMS Use Case Examples

3.1 Configuring a Device for an Untrusted Manufacturing Environment

This example will show how to order a custom part that is secure from the moment it leaves Silicon Labs. It has secure boot, secure debug lock, and encrypted upgrades enabled so that an untrusted contract manufacturer cannot access the debug port or upload unsigned and/or unencrypted applications.

This example uses an EFR32MG21B, which is a Secure Vault High part. Secure Vault Base or Mid parts do not have the same customization options, so some sections of this example will not be applicable to such devices.
3.1.1 CPMS

This section provides detailed information on starting a new custom part in CPMS and configuring the debug lock and Secure Boot.

1. In a browser, open CPMS at https://cpms.silabs.com/login.
2. Log in using your www.silabs.com account credentials.
3. Click "Create a new Custom Part":

   a. Part: Select any Secure Vault Mid or High part. This example used "EFR32MG21B010F1024IM32-B".

   b. Name: Enter "Example-1". This name will be used within CPMS to help differentiate between custom devices.

   c. Estimated Product Order Volume: Select any of the options.

   d. Estimated First Volume Order Time: Select any of the options.
4. Click "Customize". This takes you to the part customization page. Change the following configurations (configurations not listed can be left as the default):

a. **Debug Lock**: Select "Secure".

![Debug Lock Configuration](image)

b. **Configure Secure Boot, Flash Lock, and Tamper Settings**: On. Turn off "Require Verify Certificate before secure boot", since this example will not use certificates.

![Secure Boot Configuration](image)

c. **Before we can enter the keys and images, we need to generate them. This will be covered in the following sections.**
3.1.2 Generating the Application

Follow the instructions below to generate and configure an application:

1. Open "Simplicity Studio".
2. In the Launcher view, click "EXAMPLE PROJECTS & DEMOS".
3. Search for "blink", and select the Platform - Blink Bare-metal project.
4. Click "Finish".
5. There should now be a blink_baremetal project open in the Simplicity IDE view. Open blink_baremetal.slcp.

6. Click on the "SOFTWARE COMPONENTS" tab.
7. In the Search bar, search for "bootloader".
8. Click on "Platform > Bootloader > Bootloader Application Interface", and click "Install".

- Clickable link to "blink_baremetal.slcp"
9. The application image will need an `application_properties.c` file as shown below to enable secure boot. The `.cert` pointer is set to NULL to disable the application certificate option. The signatureType and signatureLocation fields are filled by Simplicity Commander when signing the application image using the convert command.

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

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

// Application properties for secure boot
const ApplicationProperties_t sl_app_properties = {
    .magic = APPLICATION_PROPERTIES_MAGIC,
    .structVersion = APPLICATION_PROPERTIES_VERSION,
    .signatureType = APPLICATION_SIGNATURE_NONE,
    .signatureLocation = 0,
    .app = {
        .type = APPLICATION_TYPE_MCU,
        .version = APP_PROPERTIES_VERSION,
        .capabilities = 0UL,
        .productId = { 0U },
    },
    .cert = NULL,
    .longTokenSectionAddress = NULL,
};
```

10. Now that the configuration is set, "Build" the project. This will generate binaries for the project.
3.1.3 Generating the Bootloader

Follow the steps below to generate and configure a bootloader.

1. Now go back to the Launcher and search for "bootloader".
2. Locate and "Create" the "Internal Storage Bootloader (single image on 1MB device)" example.
3. Open `bootloader-storage-internal-single.isc`.
4. Click on the "Plugins" tab, then select "Bootloader Core, provides API: core".
5. Click "Require encrypted firmware upgrade files" and "Enable Secure Boot".

6. At the top right, click on "Generate".
7. Now that the files have been generated, "Build" the project (if the build button is greyed out, you may need to click on the project in the Project Explorer).
3.1.4 Generating the Sign Key, the Command Key, and the OTA Decryption Key

Enabling secure boot and secure debug requires importing public keys. Ideally, these keys would be generated and managed by an HSM. This example will use Commander.

1. Create a sign key pair for secure boot:

```bash
commander util genkey --type ecc-p256 --privkey cpms-sign-priv.pem --pubkey cpms-sign-pub.pem
```

```
C:\Users\bethorel\SimplicityStudio\v5_workspace>commander util genkey --type ecc-p256 --privkey cpms-sign-priv.pem --pubkey cpms-sign-pub.pem
Generating ECC P256 key pair...
Writing private key file in PEM format to cpms-sign-priv.pem
Writing public key file in PEM format to cpms-sign-pub.pem
DONE
```

2. Create a command key pair for secure debug:

```bash
commander util genkey --type ecc-p256 --privkey cpms-cmd-priv.pem --pubkey cpms-cmd-pub.pem
```

```
C:\Users\bethorel\SimplicityStudio\v5_workspace>commander util genkey --type ecc-p256 --privkey cpms-cmd-priv.pem --pubkey cpms-cmd-pub.pem
Generating ECC P256 key pair...
Writing private key file in PEM format to cpms-cmd-priv.pem
Writing public key file in PEM format to cpms-cmd-pub.pem
DONE
```

3. Create an OTA decryption/encryption key for GBL upgrades:

```bash
commander util genkey --type aes-ccm --outfile cpms-gbl.txt
```

```
C:\Users\bethorel\SimplicityStudio\v5_workspace>commander util genkey --type aes-ccm --outfile cpms-gbl.txt
Using Windows' Cryptographic random number generator
DONE
```
### 3.1.5 Signing and Merging the Application and Bootloader Images

We now need to prepare our application and bootloader for CPMS. First, we need to sign the images. Then, since CPMS requires the firmware image to be in one file, we need to merge the signed hex files. We will do this using the Simplicity Commander command line interface.

1. Open a terminal and navigate to your Simplicity Studio workspace.
2. Sign the bootloader:

   ```bash
   commander convert "internal-storage-bootloader-single\GNU ARM v10.2.1 - Default\internal-storage-bootloader-single.hex" --secureboot --keyfile cpms-sign-priv.pem --outfile cpms-btl-signed.hex
   ``

   This will create the **cpms-btl-signed.hex** signed image file in your workspace.

   ```bash
   C:\Users\bethorel\SimplicityStudio\v5_workspace>commander convert "internal-storage-bootloader-single\GNU ARM v10.2.1 - Default\internal-storage-bootloader-single.hex" --secureboot --keyfile cpms-sign-priv.pem --outfile cpms-btl-signed.hex
   Parsing file bootloader-storage-internal-single\GNU ARM v10.2.1 - Default\bootloader-storage-internal-single.hex...
   Found Application Properties at 0x000024a8
   Writing Application Properties signature pointer to point to 0x000025e0
   Setting signature type in Application Properties: 0x00000001
   Image SHA256: ca36deb86dcb720aabe9f2d4d37c730172fe34571aedc452b52f9ef5a824264
   R = 3e8e58af660f769fe25eb626b99188b8617672352367f0ec96df6c7133b20
   S = 5c36a7b3124f320c98956b80d2ff1a1db3593bc008e11b50015e3bee463857
   Writing to cpms-btl-signed.hex...
   DONE
   ```

3. Sign the application:

   ```bash
   commander convert "blink_baremetal\GNU ARM v10.2.1 - Default\blink_baremetal.hex" --secureboot --keyfile cpms-sign-priv.pem --outfile cpms-app-signed.hex
   ``

   This will create the **cpms-app-signed.hex** signed image file in your workspace.

   ```bash
   C:\Users\bethorel\SimplicityStudio\v5_workspace>commander convert "blink_baremetal\GNU ARM v10.2.1 - Default\blink_baremetal.hex" --secureboot --keyfile cpms-sign-priv.pem --outfile cpms-app-signed.hex
   Parsing file blink_baremetal\GNU ARM v10.2.1 - Default\blink_baremetal.hex...
   Found Application Properties at 0x000061bc
   Writing Application Properties signature pointer to point to 0x000064d8
   Setting signature type in Application Properties: 0x00000001
   Image SHA256: 03bb8c3d4b7676b7a015ada8a65ba96169e0b86177548b692a385ed5b840295
   R = 0c6488ec9feff008e0744a13ca606b654c1a60108af2f5c6a0ed5a1
   S = c99de6279f50c66cd317365fd9b380d009790764a9ede0067f9e9126763844
   Writing to cpms-app-signed.hex...
   DONE
   ```
4. Merge the signed hex files:

   ```sh
   commander convert cpms-app-signed.hex cpms-btl-signed.hex -o cpms-merged.hex
   ```

   This will create **cpms-merged.hex** in your workspace.
3.1.6 Programming the Keys and Flash Memory

This section describes how to upload the public sign key and the merged signed hex file.

1. In CPMS, return to the "Standard Security Keys" section.
2. Click on the blue upload button in the "Secure Boot Key" field, then select the cpms-sign-pub.pem file.

![Standard Security Keys](image)

![Folder with cpms-sign-pub file](image)
3. Click on the blue upload button in the "Command Key" field, then select the cpms-cmd-pub.pem file.

### Standard Security Keys

#### Secure Boot Key

```
0x049fb742281f7a335c8bb2341eb23132be67ace68a1cf7e1555b17ac694cba8bb7cb0c6d99ec08c55ddf1c19ac5d4a9af6a
```

This key is used for binary authentication and/or OTA upgrade payload authentication. If you enabled secure boot, you must provide the public part of the key you used to sign your bootloader or application image here. (e.g. 0x04123456789...ABCD total 65 bytes. You can also upload a .pem or .der file)

#### Command Key

```
0x04123456789...ABCD
```

This key is used for Secure Debug Unlock or Disable Tamper command authentication. If you chose secure debug lock, you must provide the public part of your command key here. (e.g. 0x04123456789...ABCD total 65 bytes. You can also upload a .pem or .der file)

4. For the OTA Decryption Key, copy the key value (in hex) from cpms-gbl.txt into the “OTA Decryption Key” field.

```
C:\Users\bethorel\SimplicityStudio\v5_workspace>type cpms-gbl.txt
# Key randomly generated by 'util genkey'
TOKEN_MFG_SECURE_BOOTLOADER_KEY: D374A93C78C6A11D8F51D287C633165
```

```
OTA Decryption Key
0xD374A93C78C6A11D8F51D287C633165
```

This key is used for decrypting GBL payloads used for firmware upgrades. (e.g. 0x0123456789...ABCD total 16 bytes.)

5. Scroll down to the "Flash Programming" section.
6. "Firmware Type:" Select "App and Bootloader".

Flash Programming

Flash Programming involves the addition of customer specific code to a standard product. Customer code in INTEL HEX format is required.

- Fill Character
  - 0x FF
  - We will fill unused or unspecified addresses of the flash with the byte you provide here.

- Firmware Type
  - App only
  - Bootloader only
  - App and Bootloader

7. Click on "CLICK HERE OR DRAG DROP TO UPLOAD A FILE".
8. Navigate to your workspace. On Windows this will be in `C:/Users/<username>/SimplicityStudio/v5_workspace`.
9. Select `cpms-merged.hex` and click "Open". CPMS only accepts Intel Hex files for firmware images.
10. You should now be able to see the binary for the application in CPMS.

Flash Programming

Flash Programming involves the addition of customer specific code to a standard product. Customer code in INTEL HEX format is required.

Firmware

Fill Character
0x FF

We will fill unused or unspecified addresses of the flash with the byte you provide here.

Firmware Type

- App only
- Bootloader only
- App and Bootloader

11. Scroll to the top of the page, and click "PROCEED TO REVIEW".

Title
Example-1

Base Part
EFR32WG21B010F1024ID32-B

Customize Your Part

Your OPN programming data is valid and ready to be reviewing for sample programming. You can leave this page and come back at any time to complete your order. Incomplete orders are retained 30 days from last access.

12. You can now review the pricing for the custom part and the security configurations you've entered.
3.2 Importing Custom Wrapped Keys

To import custom wrapped keys into CPMS, you need four fields: value, address, auth, and metadata. The following examples will show how to get the metadata value for an asymmetric and a symmetric key.

Example #1: Importing Custom Wrapped Asymmetric Keys

1. In Simplicity Studio, in the Launcher view click on "EXAMPLE PROJECTS & DEMOS".
2. Search for "SE Manager".
3. Create a project from the "Platform - SE Manager Digital Signature (ECDSA and EdDSA)" example.

4. CPMS will automatically wrap your key and write it into flash. To emulate that for testing, we will use the Memory System Controller to write the key into flash. To enable the MSC, first open `se_manager_signature.slcp`.
5. Open the "SOFTWARE COMPONENTS" tab.
6. Search for "msc".
7. Click on the MSC Peripheral and click "Install".

The application interface consists of functions that can be included in the customer application that will communicate with the bootloader through the `MainBootloaderDefTable_t`. This table contains function pointers to the bootloader. The 10th word of the bootloader contains a pointer to this struct, allowing any application to easily locate it. To access the
8. We will modify the "create_wrap_asymmetric_key" function of `app_se_manager_signature.c` to use our "CPMS key". Instead of generating a key, we will import our ecc key. In `app_se_manager_signature.c` line 255, replace the lines:

```
print_error_cycle(sl_se_generate_key(&cmd_ctx, &asymmetric_key_desc), &cmd_ctx);
```

with the following:

```
// YOUR KEY VALUE GOES HERE:
static uint8_t user_key[64] = {
    0x79, 0x7D, 0x86, 0x83, 0x5B, 0xA3, 0x03, 0xA5,
    0xEE, 0x09, 0xAB, 0x5E, 0x7E, 0xB1, 0x2D, 0xC3,
    0x92, 0xFC, 0xCE, 0xDC, 0xD0, 0x2A, 0x80, 0xF7,
    0x56, 0x5E, 0x73, 0x30, 0x86, 0x3D, 0xAE, 0xD5,
    0xDD, 0x8A, 0x84, 0xA2, 0x87, 0x0F, 0xCC, 0x2B,
    0x70, 0x66, 0xAE, 0xE0, 0x88, 0x44, 0x2C, 0xCC,
    0xCD, 0x53, 0xCE, 0x9D, 0x26, 0xBB, 0xB3, 0x04,
    0xA8, 0xB7, 0xB9, 0xE5, 0x20, 0x43, 0x62, 0xAE
};

sl_se_key_descriptor_t plaintext_desc = {
    .type = key_type,
    .flags = SL_SE_KEY_FLAGASYMMETRIC_BUFFER_HAS_PRIVATE_KEY |
                SL_SE_KEY_FLAGASYMMETRIC_SIGNING_ONLY,
    .storage.method = SL_SE_KEY_STORAGEEXTERNALPLAINTEXT,
    .storage.location.buffer.pointer = user_key,
    .storage.location.buffer.size = 64
};
if (sl_se_import_key(&cmd_ctx, &plaintext_desc, &asymmetric_key_desc) != SL_STATUS_OK)
    return SL_STATUS_FAIL;
```

This code will import your key into the Secure Engine, wrap it, then store the wrapped key to the `asymmetric_key_buf` that `asymmetric_key_desc.storage.location.buffer.pointer` is pointing to.
9. Next, we need to write the wrapped key blob into flash. Add the following lines to `create_wrap_asymmetric_key`:

```c
// YOUR KEY ADDRESS GOES HERE:
unsigned int wrapped_key_address = 0x00080000;
printf("\nWriting key into flash at 0x%08x...\n", wrapped_key_address);

// Clear out the old wrapped key
MSC_ErasePage((uint32_t*)wrapped_key_address);

// Flash the new wrapped key
MSC_WriteWord((uint32_t*)wrapped_key_address, asymmetric_key_buf, sizeof(asymmetric_key_buf));

// Update the key descriptor to point to the key in flash
asymmetric_key_desc.storage.location.buffer.pointer = (uint8_t*)wrapped_key_address;
```

10. Next, we'll print out the keyspec that we need for CPMS. Add the following lines to `create_wrap_asymmetric_key`:

```c
unsigned int keyspec;
if (sli_se_key_to_keyspec(&asymmetric_key_desc, &keyspec) != SL_STATUS_OK)
    return SL_STATUS_FAIL;
printf("\nKeyspec: 0x%08x\n", keyspec);
return SL_STATUS_OK;
```
11. Keys imported using CPMS use a different bus master than the CPU, so the key descriptor needs to be updated. In `create_wrap_symmetric_key`, edit the `symmetric_key_desc.flags` field to remove `SL_SE_FLAGASYMMETRIC_BUFFER_HAS_PUBLIC_KEY` and add `SL_SE_KEY_FLAG_ALLOW_ANY_ACCESS` (line 229):

```c
asymmetric_key_desc.flags = SL_SE_KEY_FLAGASYMMETRIC_BUFFER_HAS_PRIVATE_KEY |
| SL_SE_KEY_FLAGASYMMETRIC_SIGNING_ONLY |
| SL_SE_KEY_FLAGNON_EXPORTABLE |
| SL_SE_KEY_FLAGALLOW_ANY_ACCESS; |
```

12. Build the project.

13. Flash to the target device.

14. In the "Debug Adapters" window, right click on the adapter for your device and click "Launch Console . . ."
15. Click on the "Serial 1" tab, then send "Enter" to start the console.

16. Reset the device. The program will first ask which type of key you want to use: plaintext, wrapped, or volatile. Type a "Space" then "Enter" to select the second option, "wrapped".

17. Type "Enter" four more times and you will see the keyspec printed to the console. When entering a custom wrapped key into CPMS, this value is the "Key Metadata" value.

18. Now that we have the key wrapped and stored in flash, we want to see that the program can use it without having the plaintext key anywhere in the application. Go back to `app_se_manager_signature.c` and comment out lines 255 to 278 and lines 283 to 289.
19. Now the application simply sets up the key descriptor to point to where we wrote the wrapped key in flash, without knowing the value of the key.

20. Repeat steps 12 to 17 to verify that the wrapped key can still be used. Note that if the flash is erased (by a commander device unlock command, for instance), this application will fail - it needs the wrapped key to be stored in flash by a previous process.
Example #2: Importing Custom Wrapped Symmetric Keys

1. In Simplicity Studio, in the Launcher view click on "EXAMPLE PROJECTS & DEMOS".
2. Search for "SE Manager".
3. Create a project from the "Platform - SE Manager Block Cipher" example:

   EFR32xG21B 2.4 GHz 10 dBm RB, WSTK Mainboard (ID: 000440169815)

   Run a pre-compiled demo or create a new project based on a software example.

   Options:
   - SE Manager
   - Demos
   - Example Projects
   - What are Demo and Example Projects?

   Technology Type
   - Amazon (6)
   - Bluetooth (6)
   - Bluetooth Mesh (6)
   - Clear Filter

   16 resources found

   - Platform - SE Manager Asymmetric Key Handling
   - Platform - SE Manager Digital Signature (ECDSA and EdDSA)
   - Platform - SE Manager Hash

   View Project Documentation

4. CPMS will automatically wrap your key and write it into flash. To emulate that for testing, we will use the Memory System Controller to write the key into flash. To enable the MSC, first open se_manager_block_cipher.slcp.
5. Open the "SOFTWARE COMPONENTS" tab.
6. Search for "msc".
7. Click on the MSC Peripheral and click "Install".

MSC - Memory System Controller

Description
Flash controller (MSC) peripheral API.

Quality
PRODUCTION
8. We will modify the "create_wrap_symmetric_key" function of app_se_manager_block_cipher.c to use our "CPMS key". Instead of generating a key, we will import our aes key. In app_se_manager_block_cipher.c line 259, replace the lines:

```c
print_error_cycle(sl_se_generate_key(&cmd_ctx, &symmetric_key_desc),
                 &cmd_ctx);
```

with the following:

```c
// YOUR KEY VALUE GOES HERE:
static uint8_t user_key[16] =
    { 0x70, 0xF4, 0x82, 0x4E, 0x49, 0xBD, 0x97, 0xAB,
        0x65, 0x65, 0x32, 0x22, 0xA0, 0x70, 0xB5, 0x16
    };

sl_se_key_descriptor_t plaintext_desc = {
    .type = SL_SE_KEY_TYPE_AES_128,
    .flags = 0,
    .storage.method = SL_SE_KEY_STORAGE_EXTERNAL_PLAINTEXT,
    .storage.location.buffer.pointer = user_key,
    .storage.location.buffer.size = 16
};

if (sl_se_import_key(&cmd_ctx, &plaintext_desc, &symmetric_key_desc) != SL_STATUS_OK)
    return SL_STATUS_FAIL;
```

This code will import your key into the Secure Engine, wrap it, then store the wrapped key to the `symmetric_key_buf` that `symmetric_key_desc.storage.location.buffer.pointer` is pointing to.
9. Next, we need to write the wrapped key blob into flash. Add the following lines to `create_wrap_symmetric_key`:

```c
// YOUR KEY ADDRESS GOES HERE:
unsigned int wrapped_key_address = 0x00080000;
printf("Writing key into flash at 0x%08x...\n", wrapped_key_address);

// Clear out the old wrapped key
MSC_ErasePage((uint32_t*)wrapped_key_address);

// Flash the new wrapped key
MSC_WriteWord((uint32_t*)wrapped_key_address, symmetric_key_buf, sizeof(symmetric_key_buf));

// Update the key descriptor to point to the key in flash
symmetric_key_desc.storage.location.buffer.pointer = (uint8_t*)wrapped_key_address;
```

10. Next, we'll print out the keyspec that we need for CPMS. Add the following lines to `create_wrap_symmetric_key`:

```c
unsigned int keyspec;
if (sli_se_key_to_keyspec(&symmetric_key_desc, &keyspec) != SL_STATUS_OK)
    return SL_STATUS_FAIL;
printf("\nKeyspec: 0x%08x\n", keyspec);
return SL_STATUS_OK;
```
11. Keys imported using CPMS use a different bus master than the CPU, so the key descriptor needs to be updated. In `create_wrap_symmetric_key`, edit the `symmetric_key_desc.flags` field to include SL_SE_KEY_FLAG_ALLOW_ANY_ACCESS (line 247):

```c
symmetric_key_desc.flags = SL_SE_KEY_FLAG_NON_EXPORTABLE | SL_SE_KEY_FLAG_ALLOW_ANY_ACCESS;
```

12. Build the project.

13. Flash to the target device.
14. In the "Debug Adapters" window, right click on the adapter for your device and click "Launch Console ...

15. Click on the Serial 1 tab, then reset the device. The program will first ask which type of key you want to use: plaintext, wrapped, or volatile. Type a Space, then "Enter" to select the second option, "wrapped".

16. Type "Enter" once more, and you will see the keyspec printed to the console. When entering a custom wrapped key into CPMS, this value is the "Key Metadata" value.
17. Type "Enter" two more times to verify that the key can be used without error. Note that if you type "Enter" after this, the program will try to use that key as a ChaCha20-Poly1305 key, and it will fail.

18. Now that we have the key wrapped and stored in flash, we want to see that the program can use it without having the plaintext key anywhere in the application. Go back to `app_se_manager_block_cipher.c` and comment out lines 259 to 275 and lines 280 to 286.

```c
259 } // YOUR KEY VALUE GOES HERE:
260 static uint8_t user_key[16] =
261 { 0x70, 0x44, 0x82, 0x0e, 0x89, 0x80, 0x97, 0x08, 0x65, 0x05, 0x32, 0x22, 0x40, 0x70, 0x85, 0x16
262 }
263 }
264 // sl_se_key_descriptor_t plaintext_desc = {
265 .type = SL_SE_KEY_TYPE_AES_128,
266 .flags = 0,
267 .storage_method = SL_SE_KEY_STORAGE_EXTERNAL_PLAINTEXT,
268 .storage.location.buffer.size = 16
269 }
270 // if (sl_se_import_key(&ctx, &plaintext_desc, &symmetric_key_desc) != SL_STATUS_OK)
271 return SL_STATUS_FAIL;
272
273 // YOUR KEY ADDRESS GOES HERE:
274 unsigned int wrapped_key_address = 0x00000000;
275
276 // print("Writing key into flash at 0x008
277 // Open the old wrapped key
278 // MSCP_erasePage(wrapped_key_address);
279 // Flash the new wrapped key
280 // MSCP_writeWord(wrapped_key_address, symmetric_key_buf, sizeof(symmetric_key_buf));
281 // Update the key descriptor to point to the key in flash
282 symmetric_key_desc.storage.location.buffer_pointer = (uint8_t*)wrapped_key_address;
```

19. Now the application simply sets up the key descriptor to point to where we wrote the wrapped key in flash, without knowing the value of the key.
Repeat steps 11 to 15 to verify that the wrapped key can still be used. Note that if the flash is erased (by a commander device unlock command, for instance), this application will fail - it needs the wrapped key to be stored in flash by a previous process.
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

Disclaimer
Silicon Labs intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Labs products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and “Typical” parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Labs reserves the right to make changes without further notice to the product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Without prior notification, Silicon Labs may update product firmware during the manufacturing process for security or reliability reasons. Such changes will not alter the specifications or the performance of the product. Silicon Labs shall have no liability for the consequences of use of the information supplied in this document. This document does not imply or expressly grant any license to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any FDA Class III devices, applications for which FDA premarket approval is required or Life Support Systems without the specific written consent of Silicon Labs. A “Life Support System” is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Labs products are not designed or authorized for military applications. Silicon Labs products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons. Silicon Labs disclaims all express and implied warranties and shall not be responsible or liable for any injuries or damages related to use of a Silicon Labs product in such unauthorized applications. Note: This content may contain offensive terminology that is now obsolete. Silicon Labs is replacing these terms with inclusive language wherever possible. For more information, visit www.silabs.com/about-us/inclusive-lexicon-project

Trademark Information
Silicon Laboratories Inc., Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, EFM®, EFM32®, EFR, Ember®, Energy Micro, Energy Micro logo and combinations thereof, “the world’s most energy friendly microcontrollers”, Redpine Signals®, WiSeConnect®, n-Link, ThreadArch®, EZLink®, EZRadio®, EZRadioPRO®, Gecko®, Gecko OS, Gecko OS Studio, Precision32®, Simplicity Studio®, Telegesis, the Telegesis Logo®, USBXpress®, Zentri, the Zentri logo and Zentri DMS, Z-Wave®, and others are trademarks or registered trademarks of Silicon Labs. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. Wi-Fi is a registered trademark of the Wi-Fi Alliance. All other products or brand names mentioned herein are trademarks of their respective holders.