# UG519: Custom Part Manufacturing Service User's Guide

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 exlains 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

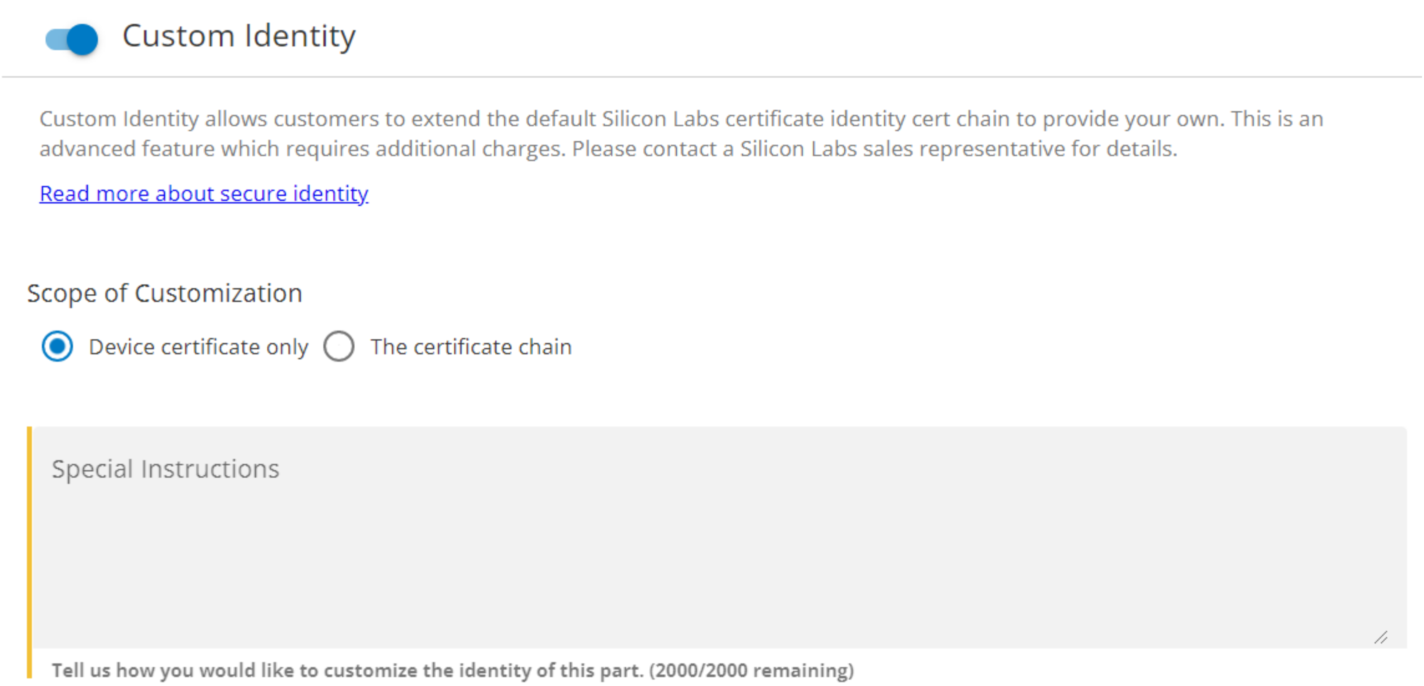
  

# 1. Custom Certificates

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:xxxxxxxxxxxxxxxx" 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.



## 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.

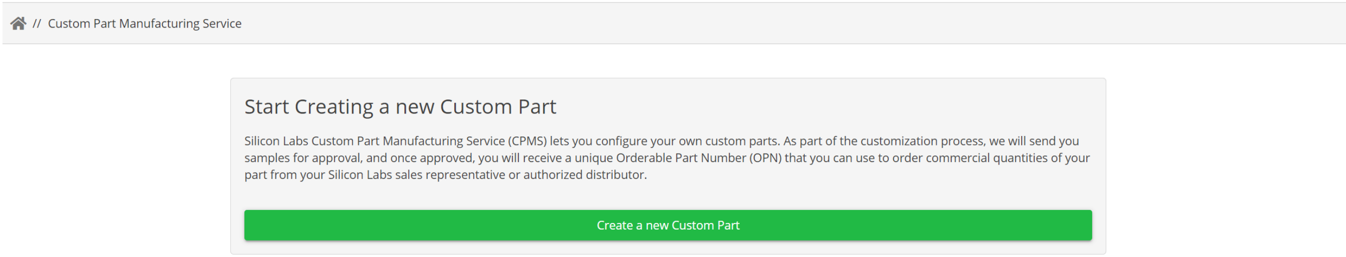
This example uses an EFR32MG21B, which is a Secure Vault High part. Secure Vault Base or Mid parts do not have the same customization options, so some sections of this example will not be applicable to such devices.

#### 3.1.1 CPMS

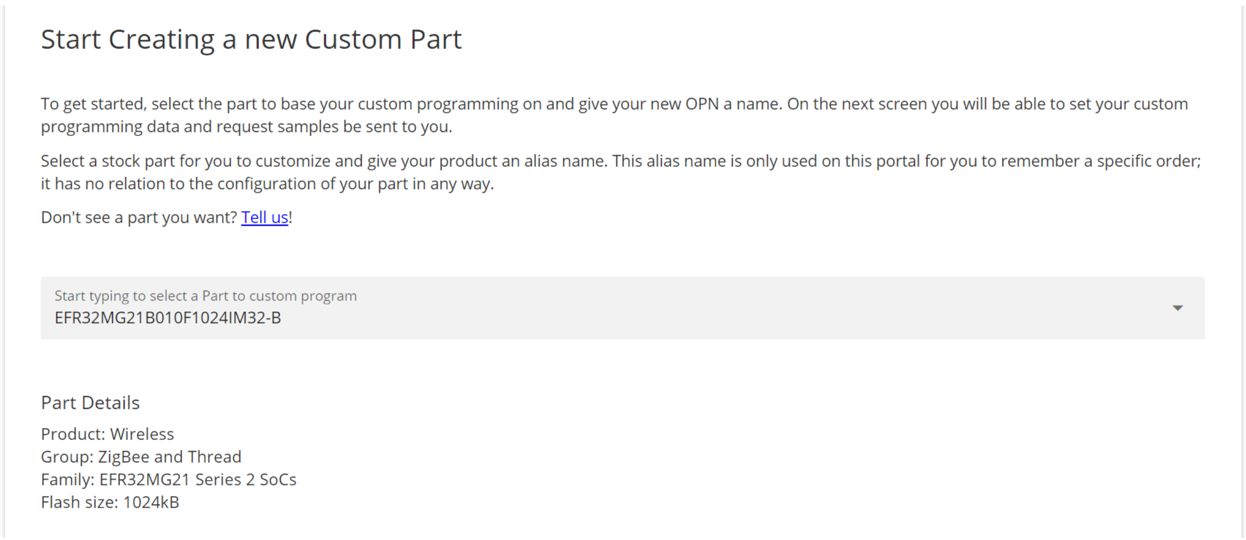
This section provides detailed information on starting a new custom part in CPMS and configuring the debug lock and Secure Boot.

1. In a browser, open CPMS at https://cpms.silabs.com/login.
2. Log in using your www.silabs.com account credentials.

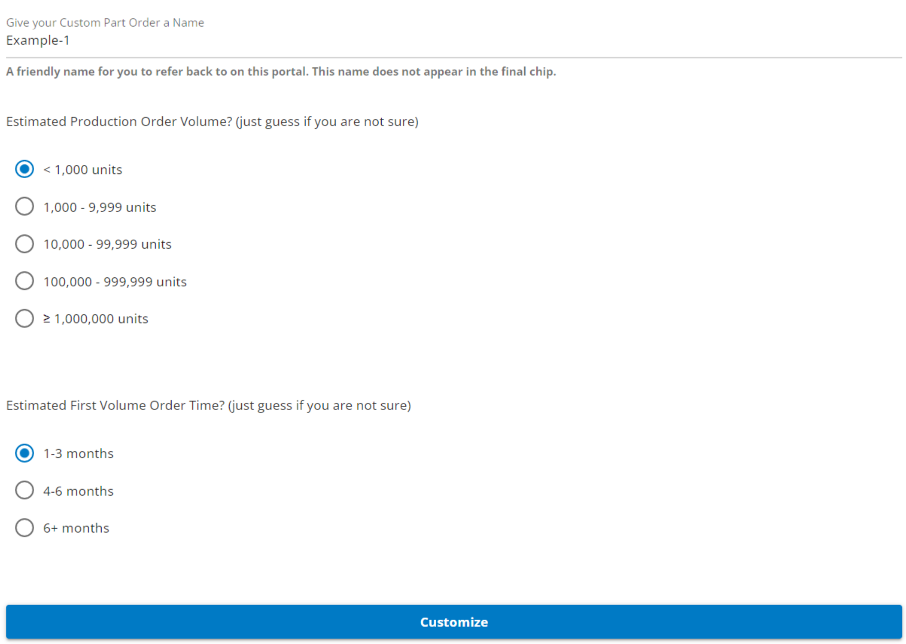
3. Click "Create a new Custom Part":



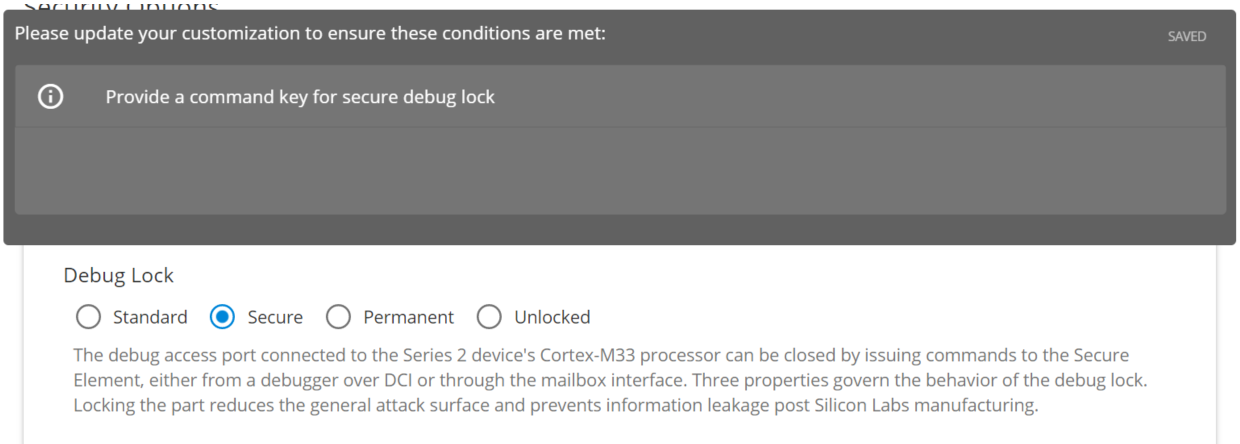

a. Part: Select any Secure Vault Mid or High part. This example used "EFR32MG21B010F1024IM32-B".



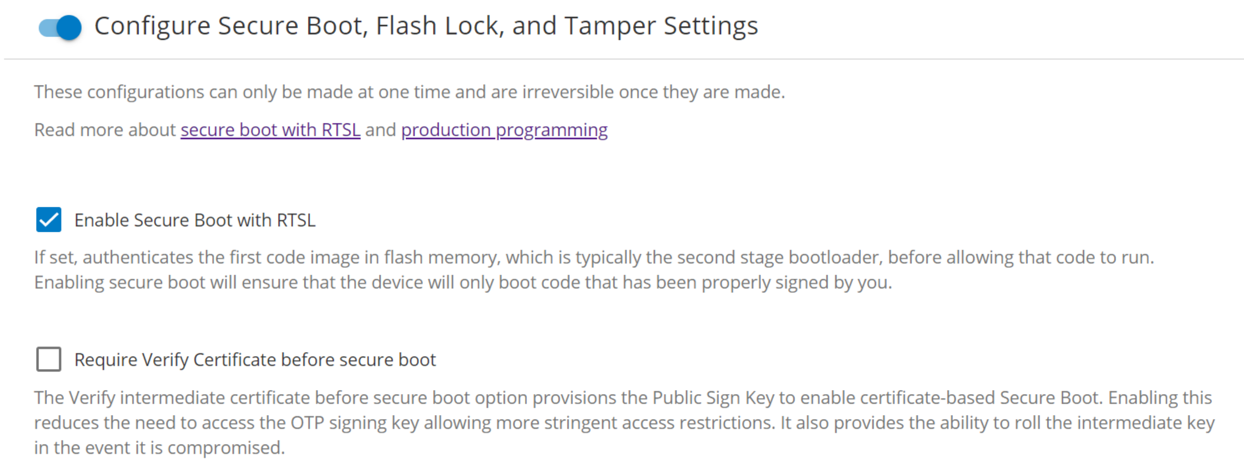

b. Name: Enter "Example-1". This name will be used within CPMS to help differentiate between custom devices.
c. Estimated Product Order Volume: Select any of the options.
d. Estimated First Volume Order Time: Select any of the options.



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".




   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.

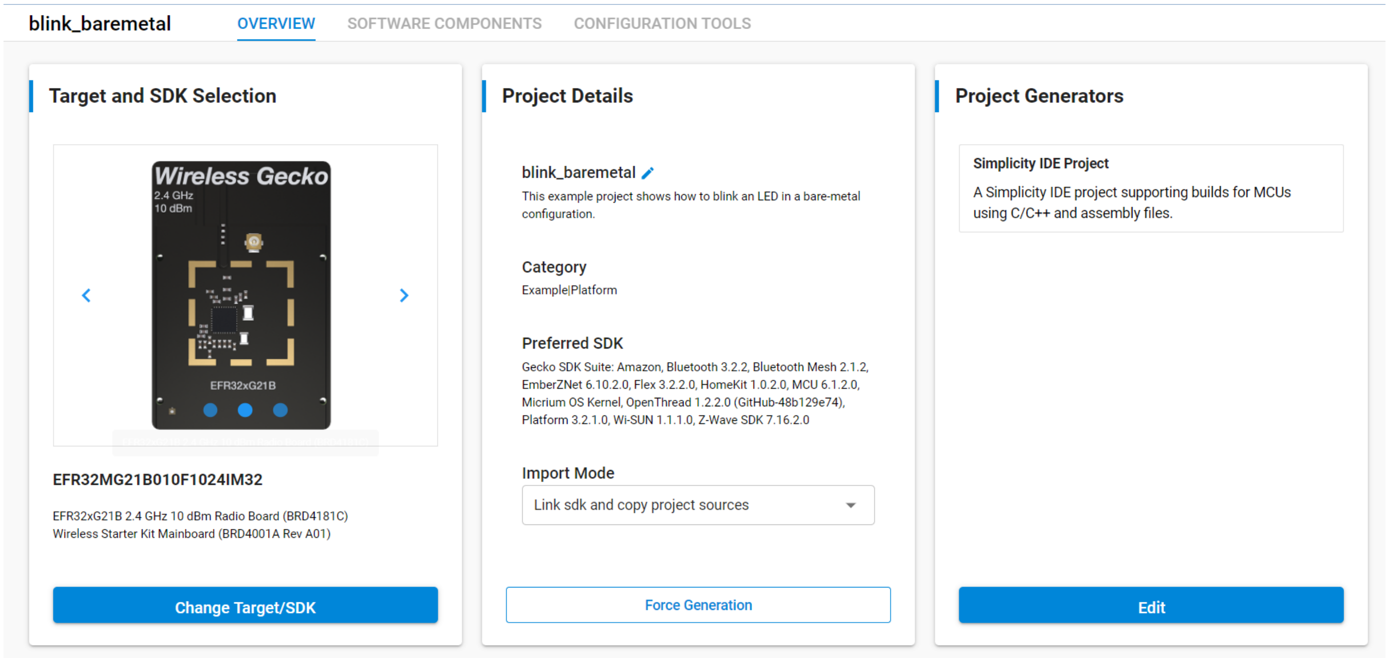



   c. Before we can enter the keys and images, we need to generate them. This will be covered in the following sections.

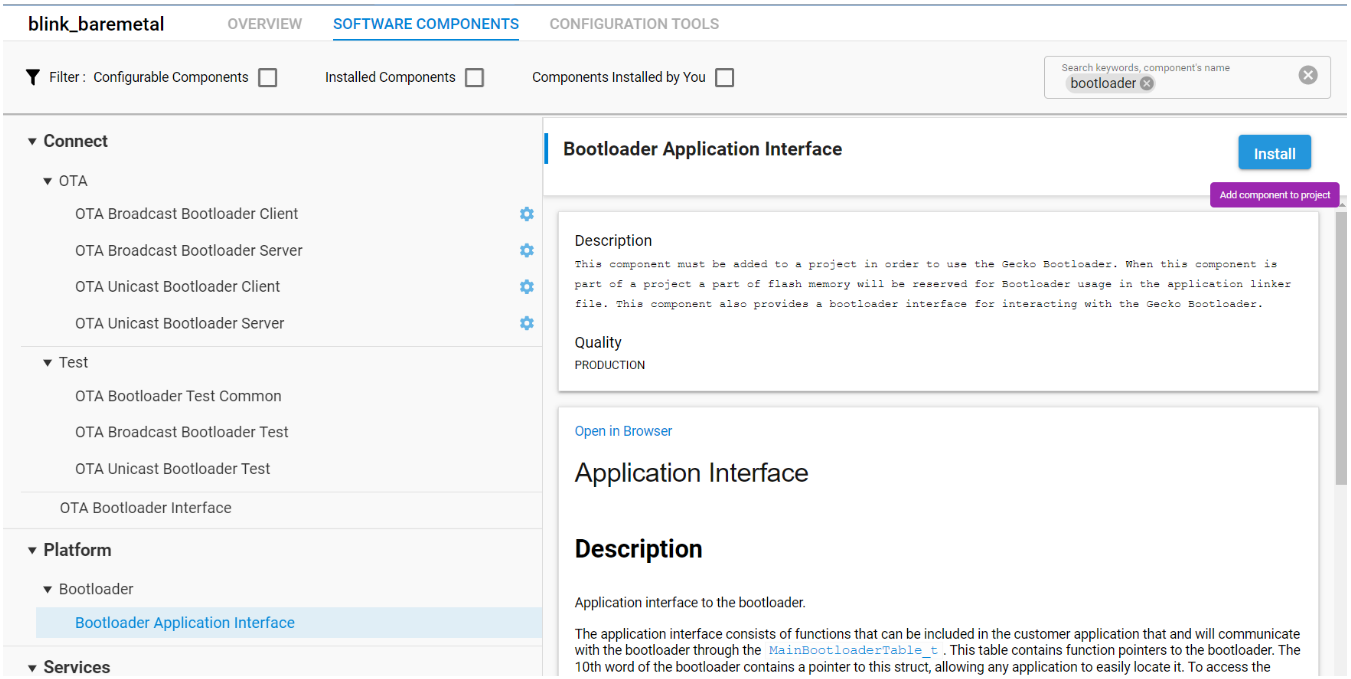
### 3.1.2 Generating the Application

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`.

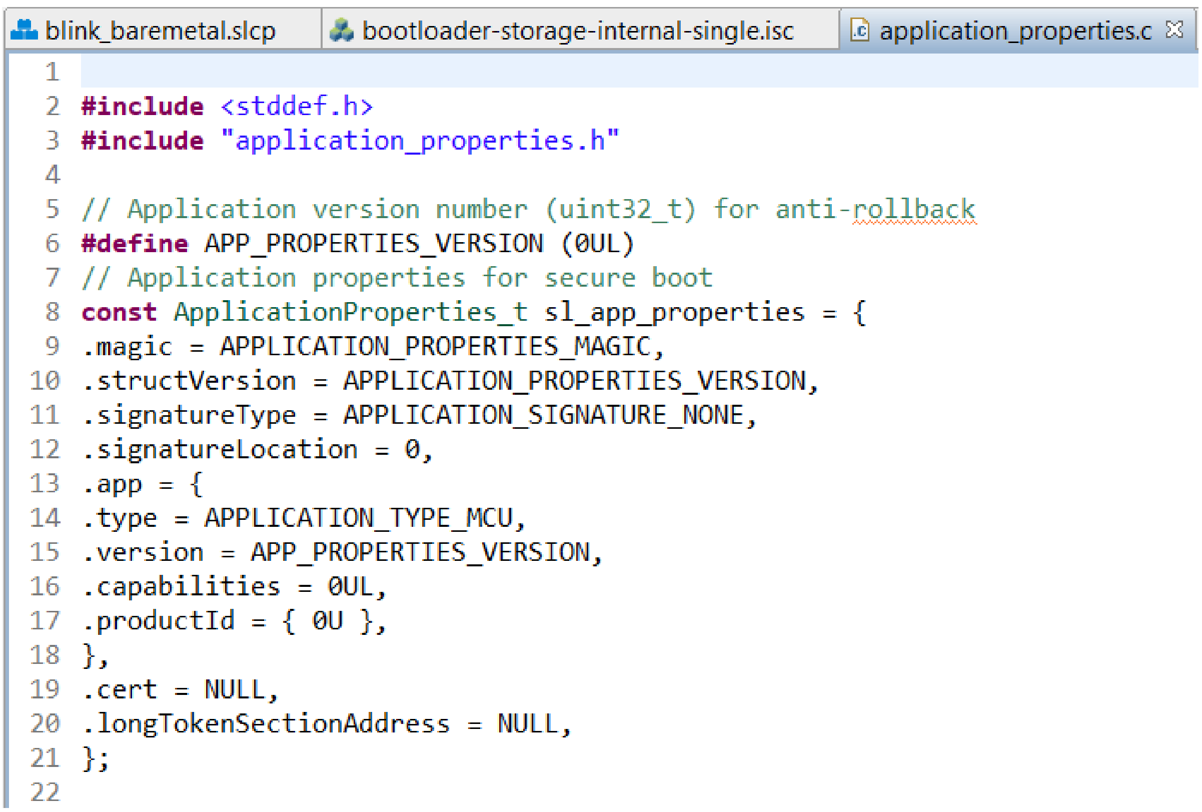



6. Click on the "SOFTWARE COMPONENTS" tab.
7. In the Search bar, search for “bootloader”.
8. Click on "Platform > Bootloader > Bootloader Application Interface", and click "Install".



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.

```
#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,
         };
```

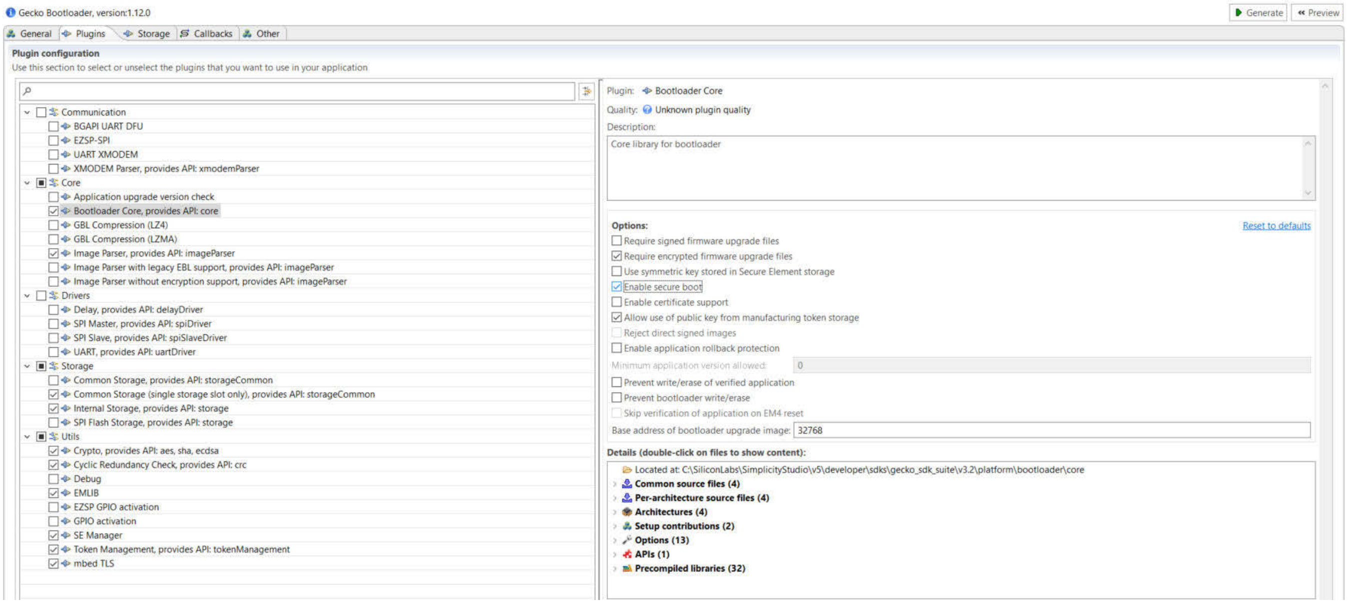



10. Now that the configuration is set, "Build" the project. This will generate binaries for the project.

### 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:

```
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:

```
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:

```
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:

```
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.

```
C:\Users\bethorel\SimplicityStudio\v5_workspace>commander convert "bootloader-sto
rage-internal-single\GNU ARM v10.2.1 - Default\bootloader-storage-internal-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\bootloa
der-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: ca36debc860cdb720aabe9fdd37dc730172fe34571aedc452b52f9ef5a824264
R = 3E8E58AF660F769FE25E9262E6899188B61716723352367F0EC96DF6C7133B20
S = 5C36A7B3124F320C9B9B56B80D2F1A1D8B3593BC008E11B50015E3BEE4638537
Writing to cpms-btl-signed.hex...
DONE
```

3. Sign the application:

```
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.

```
C:\Users\bethorel\SimplicityStudio\v5_workspace>commander convert "blink_baremeta
l\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: 030b8cdb43e7666b1a015ada8a658a96169be086177548b692a385edb5840295
R = 0C64B8EC9FEFD081EFEBF08E0744A13CA606BD654C1A6B108AF2F5C06AECD5A1
S = CA9DE6279F50C86CD317365FD98380D097D90764A9EDEFE06623FE9126763844
Writing to cpms-app-signed.hex...
DONE
```

4. Merge the signed hex files:

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

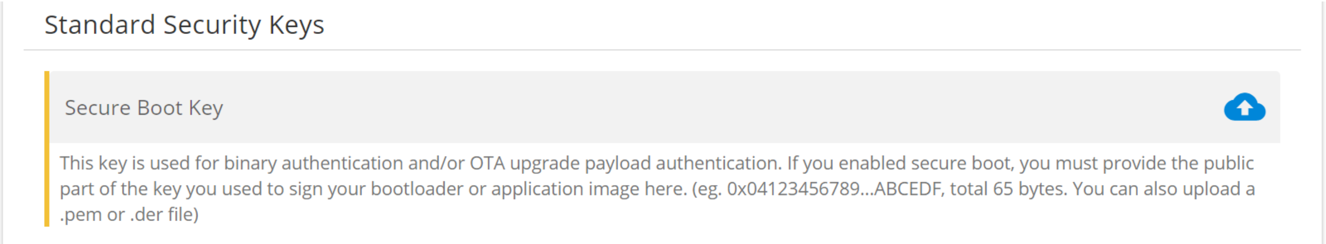
```
C:\Users\bethorel\SimplicityStudio\v5_workspace>commander convert cpms-app-signed
.hex cpms-btl-signed.hex -o cpms-merged.hex
Parsing file cpms-app-signed.hex...
Parsing file cpms-btl-signed.hex...
Writing to cpms-merged.hex...
DONE
```

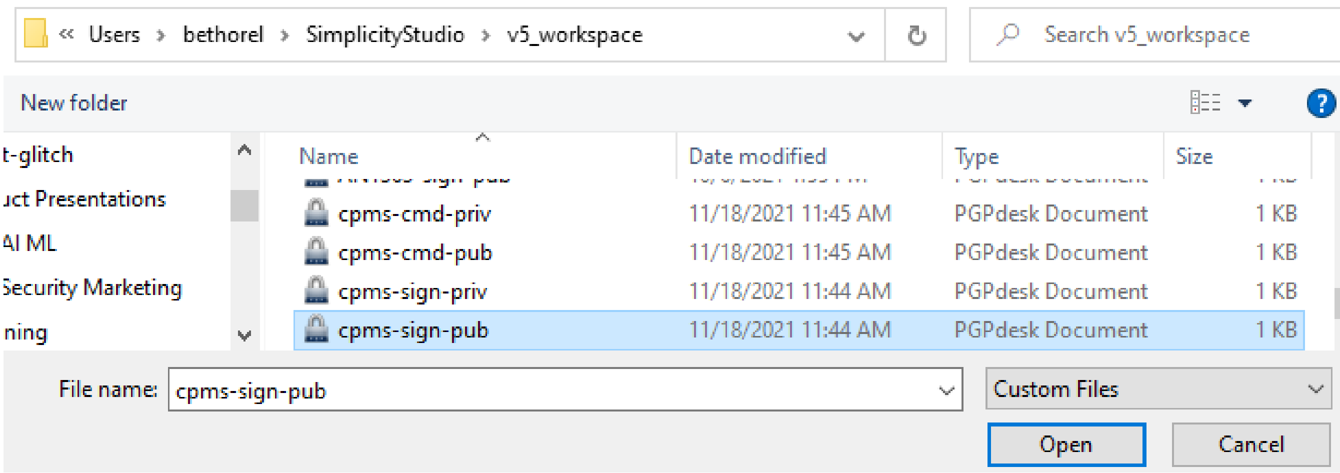
This will create **cpms-merged.hex** in your workspace.

### 3.1.6 Programming the Keys and Flash Memory

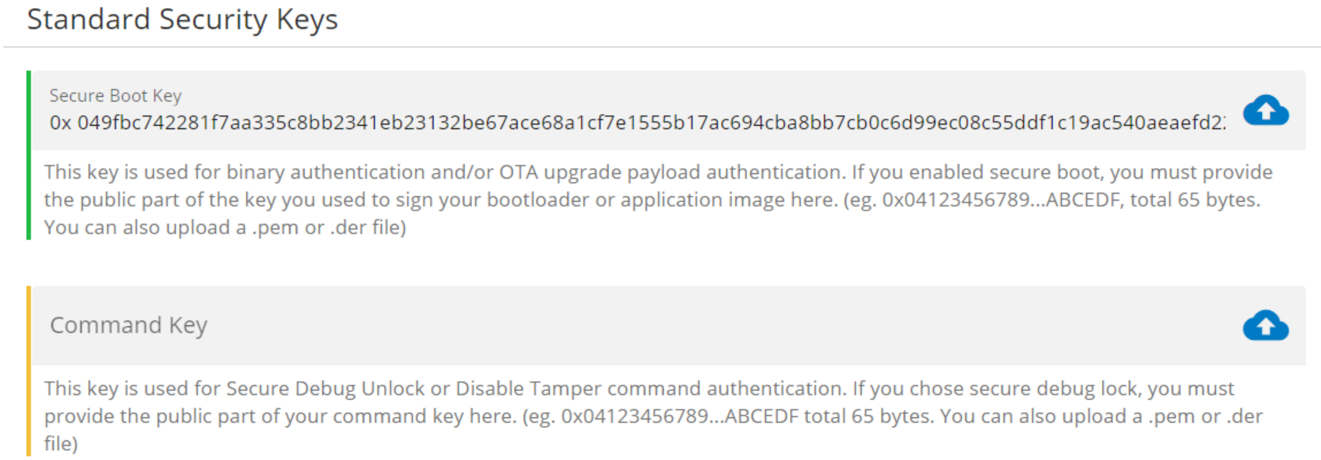
This section describes how to upload the public sign key and the merged signed hex file.

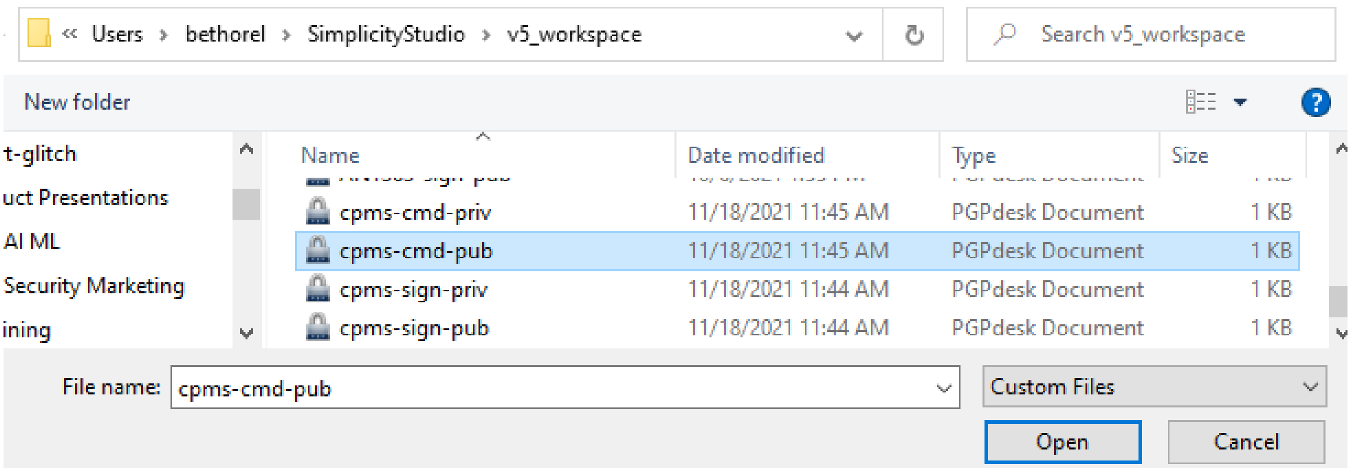
1. In CPMS, return to the "Standard Security Keys" section.
2. Click on the blue upload button in the "Secure Boot Key" field, then select the **cpms-sign-pub.pem** file.



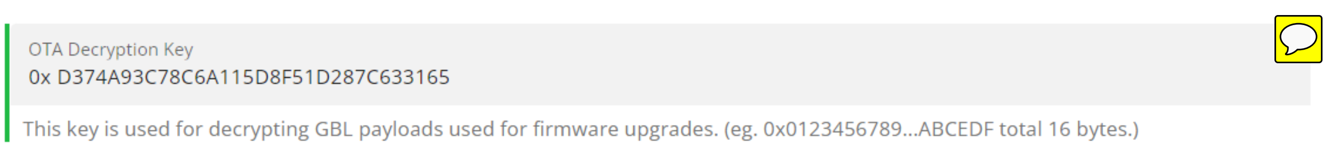
3. Click on the blue upload button in the "Command Key" field, then select the **cpms-cmd-pub.pem** file.



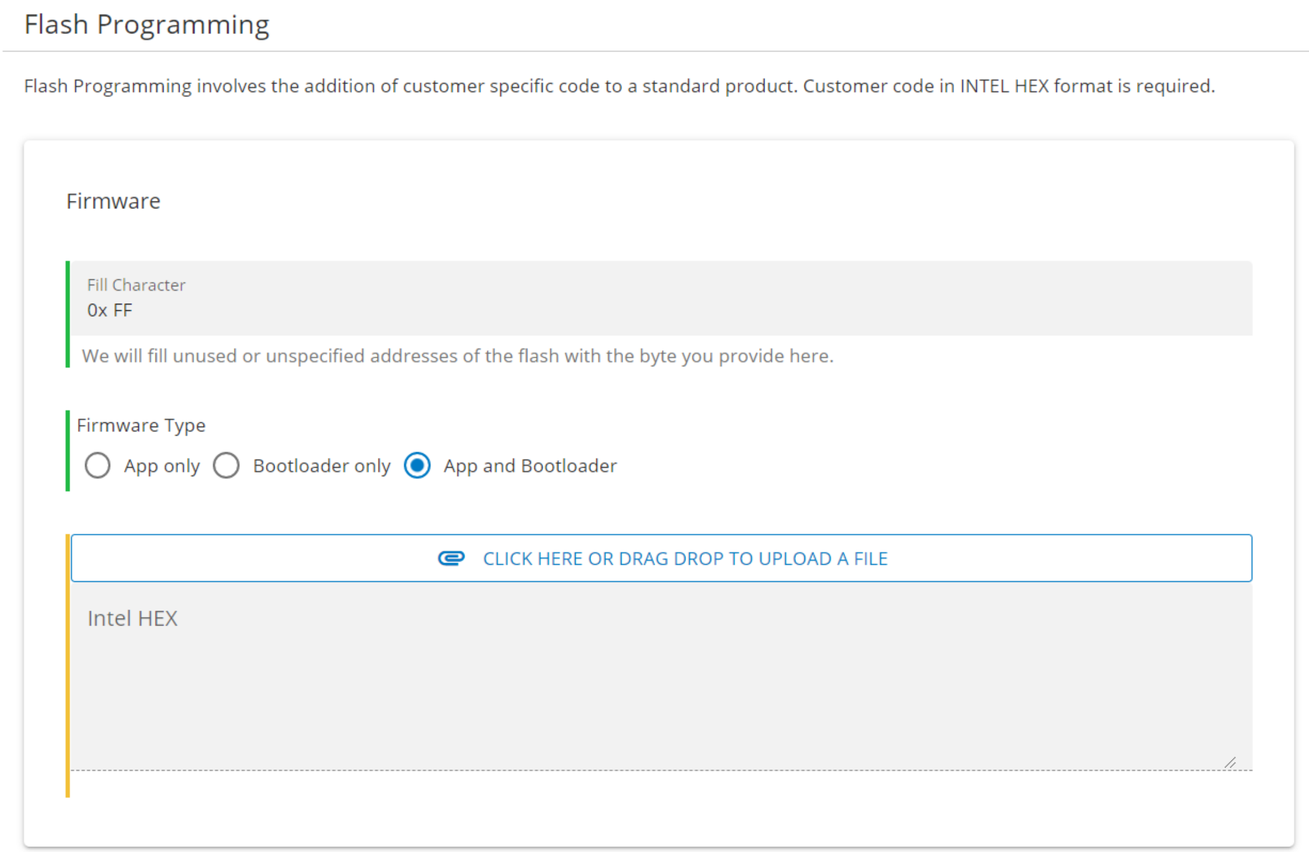

4. For the OTA Decryption Key, copy the key value (in hex) from **cpms-gbl.txt** into the "OTA Decryption Key" field.

```
C:\Users\bethorel\SimplicityStudio\v5_workspace>type cpms-gbl.txt
# Key randomly generated by 'util genkey'
TOKEN_MFG_SECURE_BOOTLOADER_KEY: D374A93C78C6A115D8F51D287C633165
```

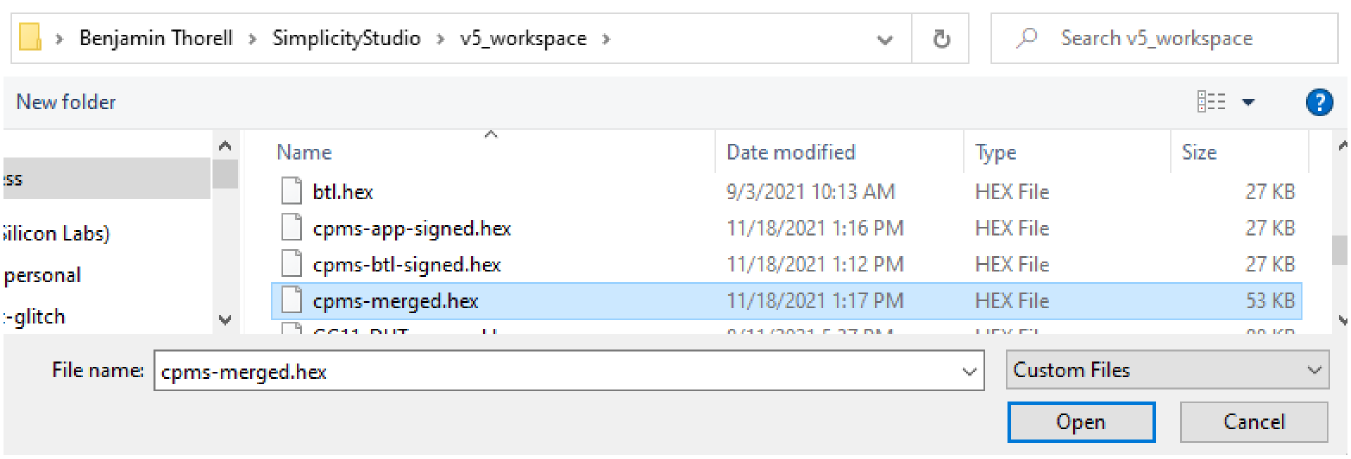



5. Scroll down to the "Flash Programming" section.

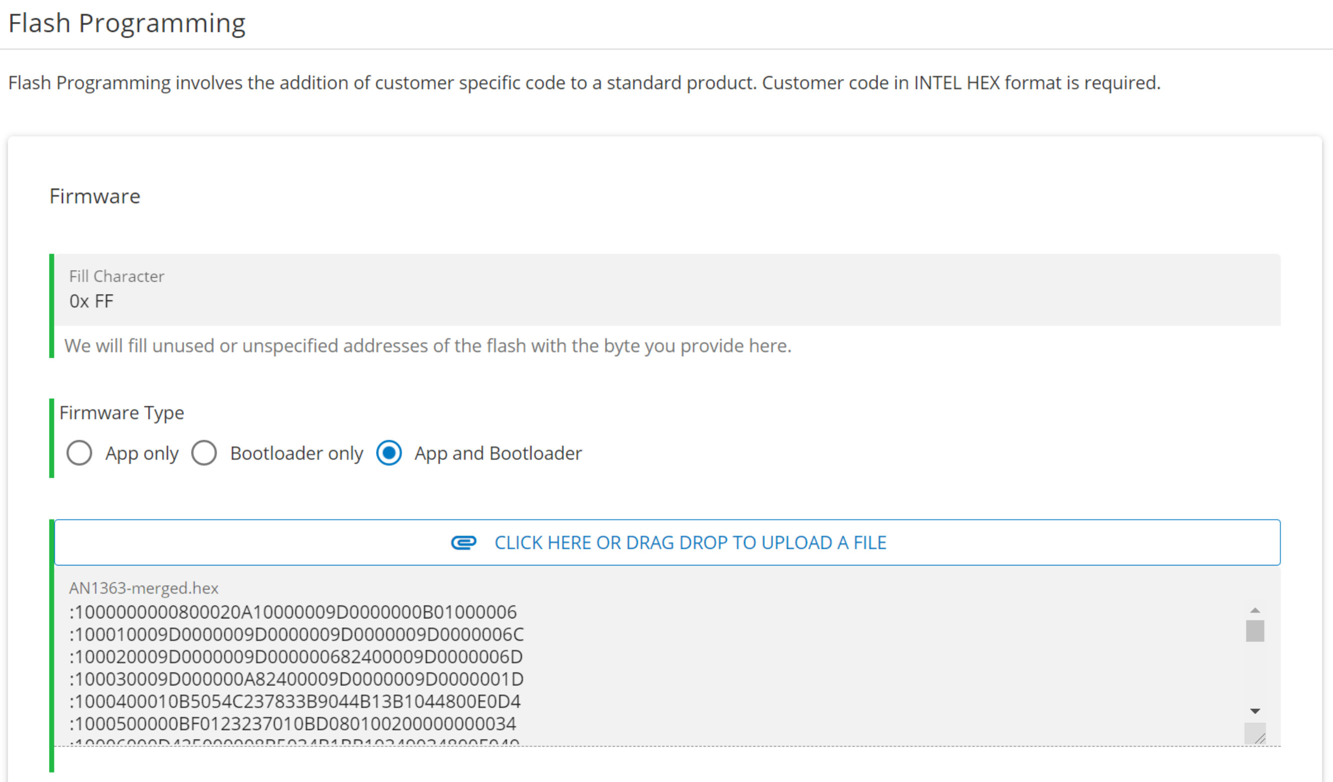
6. "Firmware Type:" Select "App and Bootloader".



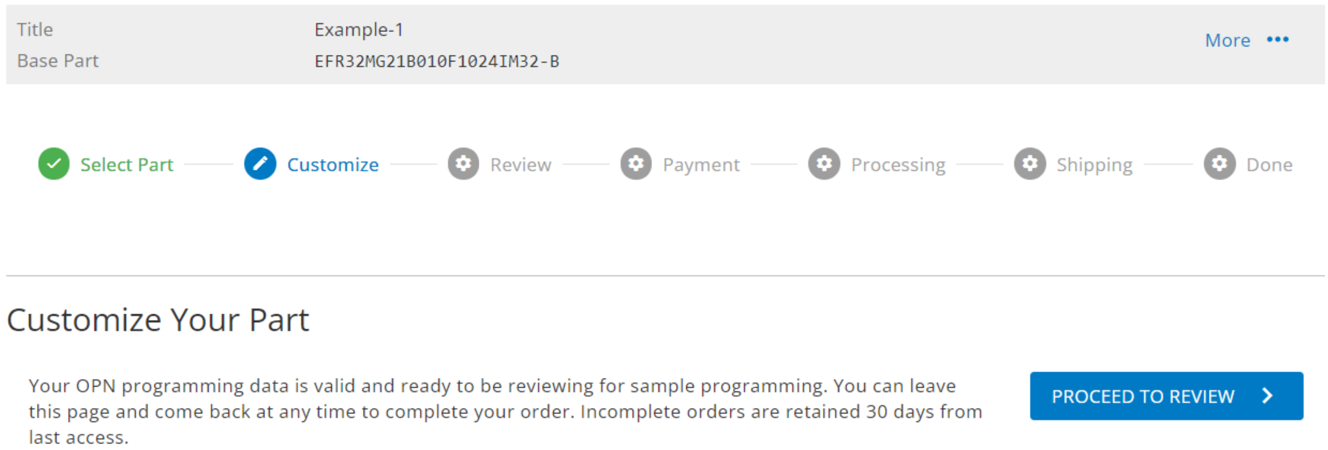

7. Click on "CLICK HERE OR DRAG DROP TO UPLOAD A FILE".
8. Navigate to your workspace. On Windows this will be in **C:/Users/<username>/SimplicityStudio/v5_workspace**.
9. Select **cpms-merged.hex** and click "Open". CPMS only accepts Intel Hex files for firmware images.



10. You should now be able to see the binary for the application in CPMS.




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



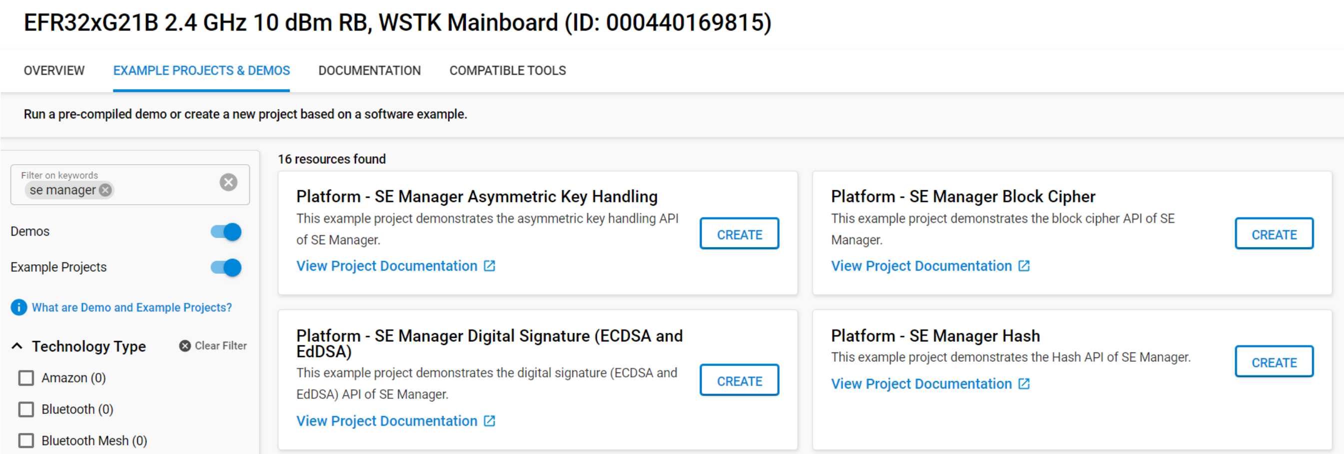

12. You can now review the pricing for the custom part and the security configurations you've entered.

### 3.2 Importing Custom Wrapped Keys

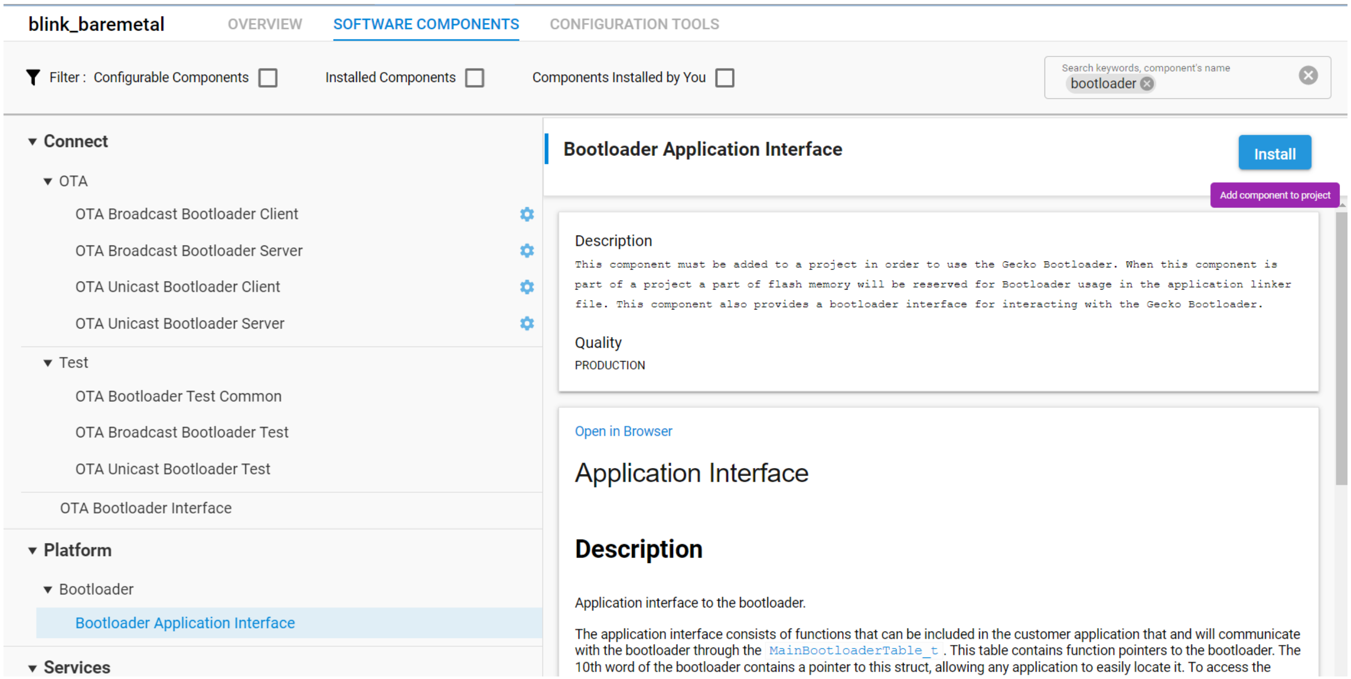
To import custom wrapped keys into CPMS, you need four fields: value, address, auth, and metadata. The following examples will show how to get the metadata value for an asymmetric and a symmetric key.

**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".



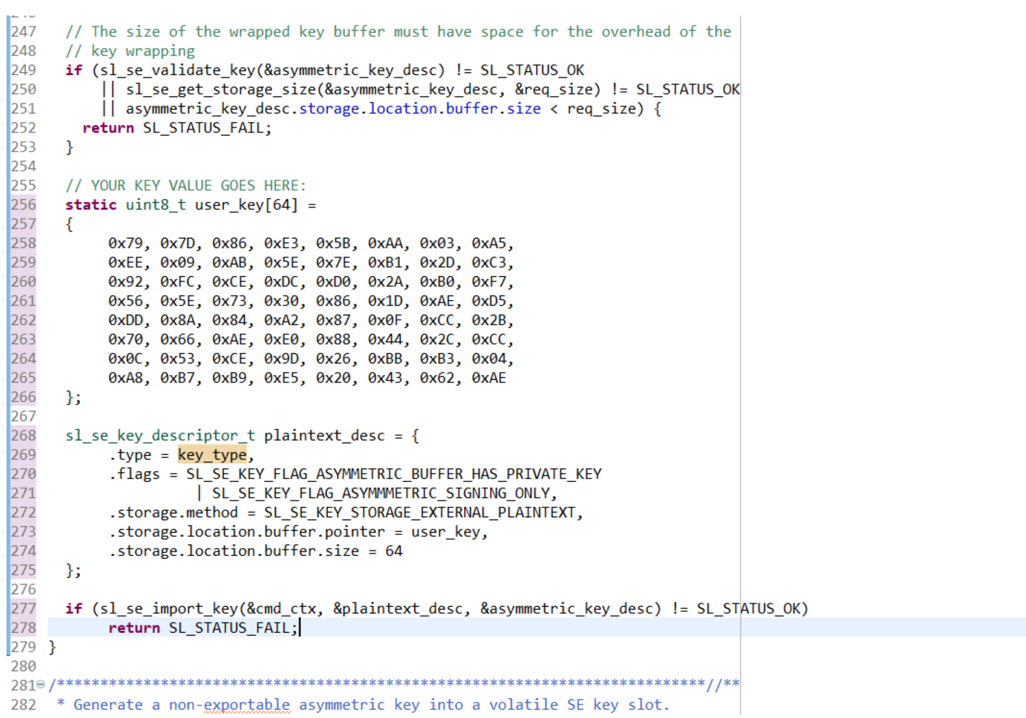
8. We will modify the "create_wrap_asymmetric_key" function of **app_se_manager_signature.c** to use our "CPMS key". Instead of generating a key, we will import our ecc key. In **app_se_manager_signature.c** line **255**, replace the lines:

```
 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, 0xE3, 0x5B, 0xAA, 0x03, 0xA5,
     0xEE, 0x09, 0xAB, 0x5E, 0x7E, 0xB1, 0x2D, 0xC3,
     0x92, 0xFC, 0xCE, 0xDC, 0xD0, 0x2A, 0xB0, 0xF7,
     0x56, 0x5E, 0x73, 0x30, 0x86, 0x1D, 0xAE, 0xD5,
     0xDD, 0x8A, 0x84, 0xA2, 0x87, 0x0F, 0xCC, 0x2B,
     0x70, 0x66, 0xAE, 0xE0, 0x88, 0x44, 0x2C, 0xCC,
     0x0C, 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_FLAG_ASYMMETRIC_BUFFER_HAS_PRIVATE_KEY
                | SL_SE_KEY_FLAG_ASYMMMETRIC_SIGNING_ONLY,
     .storage.method = SL_SE_KEY_STORAGE_EXTERNAL_PLAINTEXT,
     .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.



```
247   // The size of the wrapped key buffer must have space for the overhead of the
248   // key wrapping
249   if (sl_se_validate_key(&asymmetric_key_desc) != SL_STATUS_OK
250       || sl_se_get_storage_size(&asymmetric_key_desc, &req_size) != SL_STATUS_OK
251       || asymmetric_key_desc.storage.location.buffer.size < req_size) {
252     return SL_STATUS_FAIL;
253   }
254
255   // YOUR KEY VALUE GOES HERE:
256   static uint8_t user_key[64] =
257   {
258        0x79, 0x7D, 0x86, 0xE3, 0x5B, 0xAA, 0x03, 0xA5,
259        0xEE, 0x09, 0xAB, 0x5E, 0x7E, 0xB1, 0x2D, 0xC3,
260        0x92, 0xFC, 0xCE, 0xDC, 0xD0, 0x2A, 0xB0, 0xF7,
261        0x56, 0x5E, 0x73, 0x30, 0x86, 0x1D, 0xAE, 0xD5,
262        0xDD, 0x8A, 0x84, 0xA2, 0x87, 0x0F, 0xCC, 0x2B,
263        0x70, 0x66, 0xAE, 0xE0, 0x88, 0x44, 0x2C, 0xCC,
264        0x0C, 0x53, 0xCE, 0x9D, 0x26, 0xBB, 0xB3, 0x04,
265        0xA8, 0xB7, 0xB9, 0xE5, 0x20, 0x43, 0x62, 0xAE
266   };
267
268   sl_se_key_descriptor_t plaintext_desc = {
269        .type = key_type,
270        .flags = SL_SE_KEY_FLAG_ASYMMETRIC_BUFFER_HAS_PRIVATE_KEY
271                   | SL_SE_KEY_FLAG_ASYMMMETRIC_SIGNING_ONLY,
272        .storage.method = SL_SE_KEY_STORAGE_EXTERNAL_PLAINTEXT,
273        .storage.location.buffer.pointer = user_key,
274        .storage.location.buffer.size = 64
275   };
276
277   if (sl_se_import_key(&cmd_ctx, &plaintext_desc, &asymmetric_key_desc) != SL_STATUS_OK)
278        return SL_STATUS_FAIL;
279 }
280
281 /***************************************************************************//**
282  * Generate a non-exportable asymmetric key into a volatile SE key slot.
```

9. Next, we need to write the wrapped key blob into flash. Add the following lines to **create_wrap_asymmetric_key**:

```
  // 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;
```

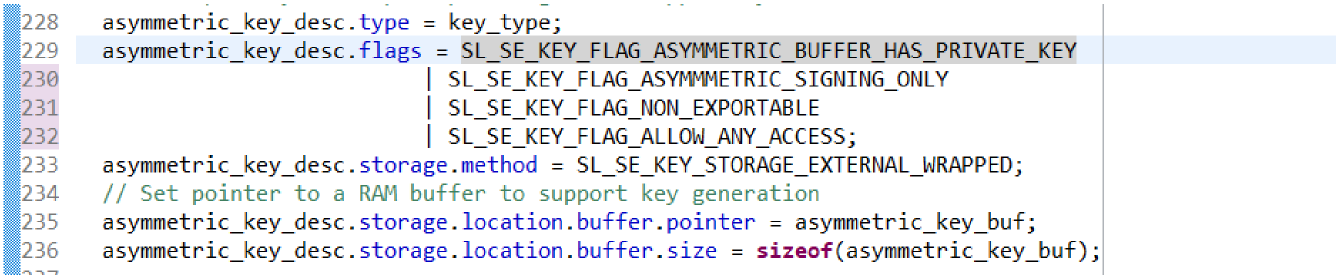
10. Next, we'll print out the keyspec that we need for CPMS. Add the following lines to **create_wrap_asymmetric_key**:

```
  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;
```

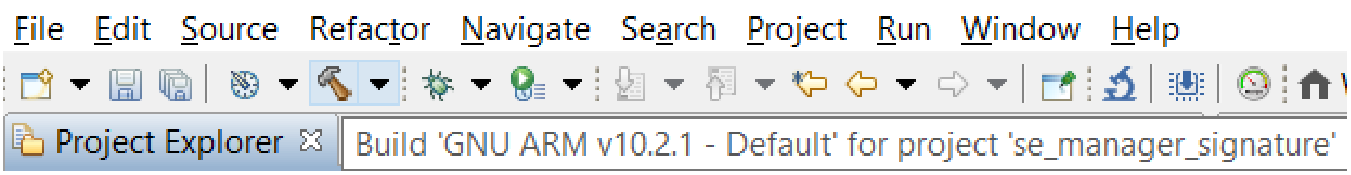
```
  };

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

  // 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;
  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_FLAG_ASYMMETRIC_BUFFER_HAS_PUBLIC_KEY and add SL_SE_KEY_FLAG_ALLOW_ANY_ACCESS (line **229**):

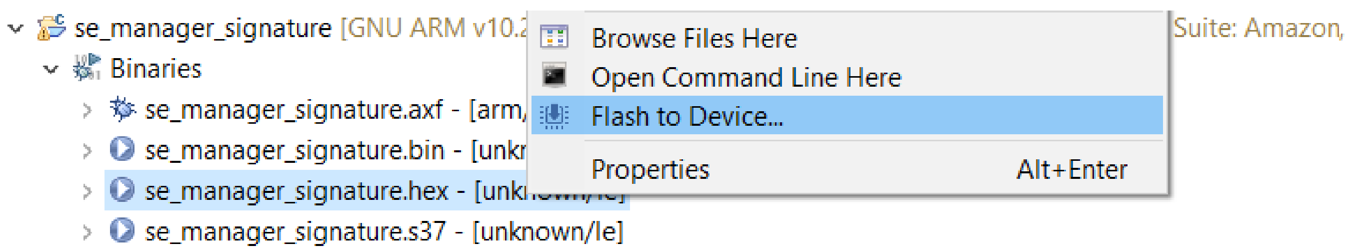
```
asymmetric_key_desc.flags = SL_SE_KEY_FLAG_ASYMMETRIC_BUFFER_HAS_PRIVATE_KEY
                | SL_SE_KEY_FLAG_ASYMMMETRIC_SIGNING_ONLY
                | SL_SE_KEY_FLAG_NON_EXPORTABLE
                | SL_SE_KEY_FLAG_ALLOW_ANY_ACCESS;
```



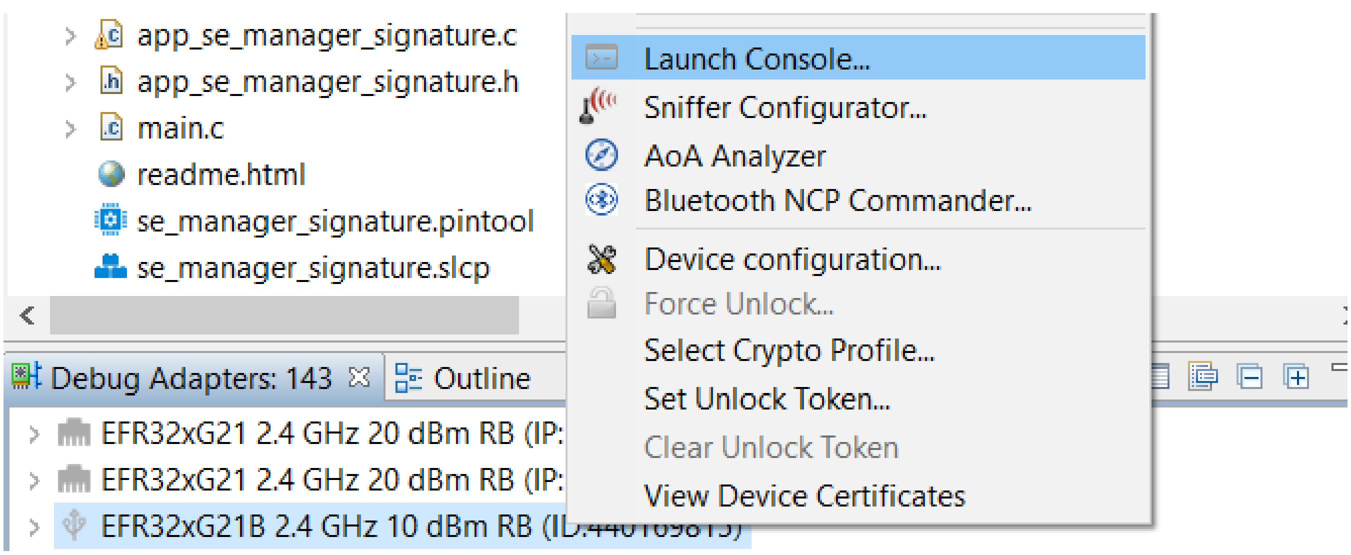

12. Build the project.




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 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".

```
Serial 0 | Serial 1 | Admin | Debug

  . Current asymmetric key algorithm is ECC Weierstrass Prime.
  + Press SPACE to select asymmetric key algorithm (ECC Weierstrass Prime/ECC EdDSA (Ed25519)), press ENTER to next option.

SE Manager Digital Signature (ECDSA and EdDSA) Example - Core running at 38000 kHz.
  . SE manager initialization... SL_STATUS_OK (cycles: 9 time: 0 us)
  + Fill 4096 bytes plain message buffer with random number... SL_STATUS_OK (cycles: 68068 time: 1791 us)

  . Current asymmetric key is a plaintext key.
  + Press SPACE to select a plaintext or wrapped or volatile key, press ENTER to next option.
  + Current asymmetric key is a wrapped key.

  . Current asymmetric key algorithm is ECC Weierstrass Prime.
  + Press SPACE to select asymmetric key algorithm (ECC Weierstrass Prime/ECC EdDSA (Ed25519)), press ENTER to next option.
```

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.

```
  . Digital signature
  + ECC Weierstrass Prime - ECC P192
  + Generate a non-exportable wrapped asymmetric key...
Writing key into flash at 0x00080000...

Keyspec: 0x8900c417
  + Sign 256 bytes message with SHA1 and wrapped private key... SL_STATUS_OK (cycles: 131246 time: 3453 us)
  + Export public key from private key... SL_STATUS_OK (cycles: 118968 time: 3130 us)
  + Verify signature with SHA1 and wrapped public key... SL_STATUS_OK (cycles: 121817 time: 3205 us)

  . Current asymmetric key is a wrapped key.
  + Press SPACE to select a plaintext or wrapped or volatile key, press ENTER to next option.
```

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.

```
252     return SL_STATUS_FAIL;
253   }
254
255 //  // YOUR KEY VALUE GOES HERE:
256 //  static uint8_t user_key[64] =
257 //  {
258 //       0x79, 0x7D, 0x86, 0xE3, 0x5B, 0xAA, 0x03, 0xA5,
259 //       0xEE, 0x09, 0xAB, 0x5E, 0x7E, 0xB1, 0x2D, 0xC3,
260 //       0x92, 0xFC, 0xCE, 0xDC, 0xD0, 0x2A, 0xB0, 0xF7,
261 //       0x56, 0x5E, 0x73, 0x30, 0x86, 0x1D, 0xAE, 0xD5,
262 //       0xDD, 0x8A, 0x84, 0xA2, 0x87, 0x0F, 0xCC, 0x2B,
263 //       0x70, 0x66, 0xAE, 0xE0, 0x88, 0x44, 0x2C, 0xCC,
264 //       0x0C, 0x53, 0xCE, 0x9D, 0x26, 0xBB, 0xB3, 0x04,
265 //       0xA8, 0xB7, 0xB9, 0xE5, 0x20, 0x43, 0x62, 0xAE
266 //  };
267 //
268 //  sl_se_key_descriptor_t plaintext_desc = {
269 //       .type = key_type,
270 //       .flags = SL_SE_KEY_FLAG_ASYMMETRIC_BUFFER_HAS_PRIVATE_KEY
271 //                 | SL_SE_KEY_FLAG_ASYMMMETRIC_SIGNING_ONLY,
272 //       .storage.method = SL_SE_KEY_STORAGE_EXTERNAL_PLAINTEXT,
273 //       .storage.location.buffer.pointer = user_key,
274 //       .storage.location.buffer.size = 64
275 //  };
276 //
277 //  if (sl_se_import_key(&cmd_ctx, &plaintext_desc, &asymmetric_key_desc) != SL_STATUS_OK)
278 //       return SL_STATUS_FAIL;
279
280   // YOUR KEY ADDRESS GOES HERE:
281   unsigned int wrapped_key_address = 0x00080000;
282
283 //  printf("\nWriting key into flash at 0x%08x...\n",  wrapped_key_address);
284 //
285 //  // Clear out the old wrapped key
286 //  MSC_ErasePage((uint32_t*)wrapped_key_address);
287 //
288 //  // Flash the new wrapped key
289 //  MSC_WriteWord((uint32_t*)wrapped_key_address, asymmetric_key_buf, sizeof(asymmetric_key_buf));
290
291   // Update the key descriptor to point to the key in flash
292   asymmetric_key_desc.storage.location.buffer.pointer = (uint8_t*)wrapped_key_address;
293   unsigned int keyspec;
```

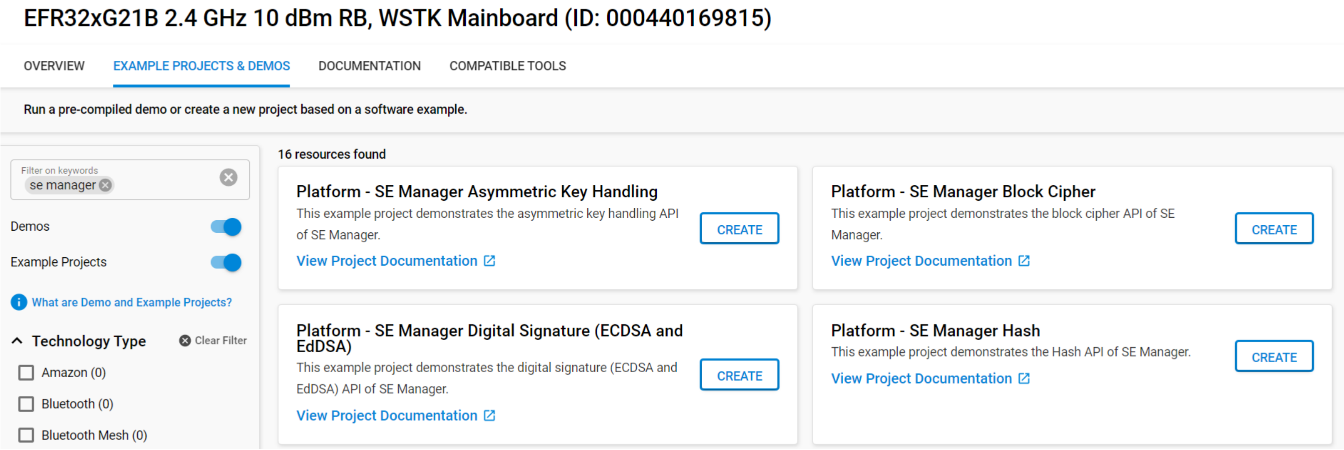
19. Now the application simply sets up the key descriptor to point to where we wrote the wrapped key in flash, without knowing the value of the key.

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.

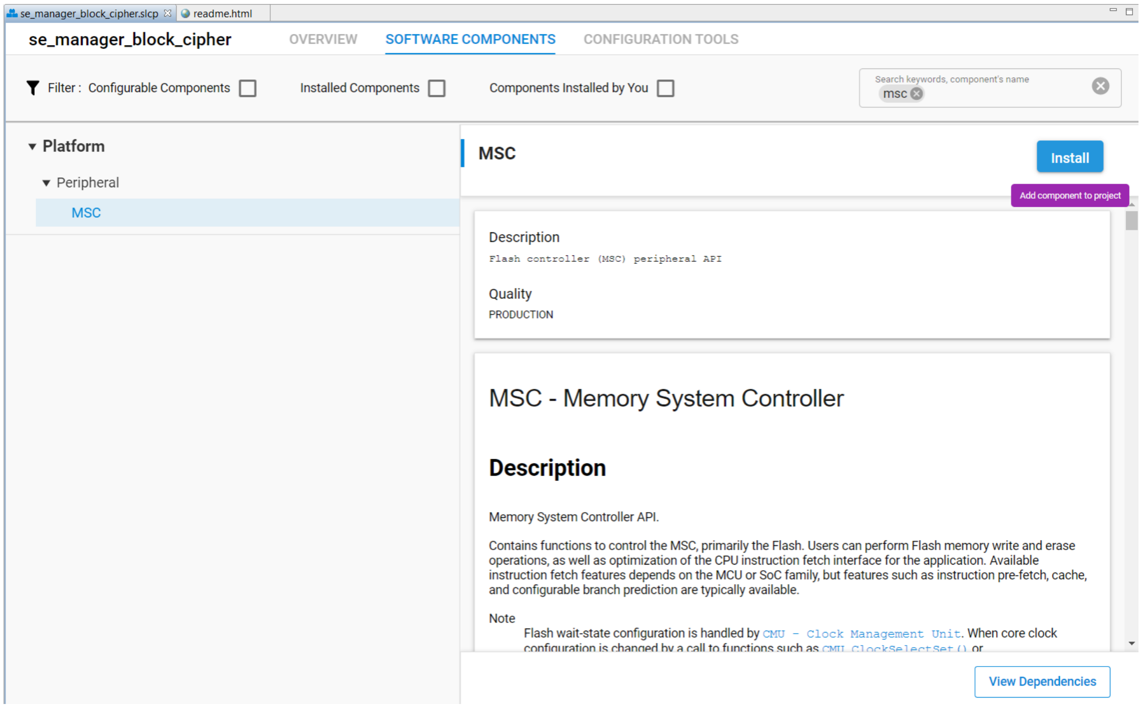
```
SE Manager Digital Signature (ECDSA and EdDSA) Example - Core running at 38000 kHz.
  . SE manager initialization... SL_STATUS_OK (cycles: 9 time: 0 us)
  + Fill 4096 bytes plain message buffer with random number... SL_STATUS_OK (cycles: 68674 time: 1807 us)

  . Current asymmetric key is a plaintext key.
  + Press SPACE to select a plaintext or wrapped or volatile key, press ENTER to next option.
  + Current asymmetric key is a wrapped key.

  . Current asymmetric key algorithm is ECC Weierstrass Prime.
  + Press SPACE to select asymmetric key algorithm (ECC Weierstrass Prime/ECC EdDSA (Ed25519)), press ENTER to next option.

  . Current ECC Weierstrass Prime key is ECC P192.
  + Press SPACE to select ECC Weierstrass Prime key (ECC P192/ECC P256/ECC P384/ECC P521/ECC Custom (secp256k1 in this example)), press ENTER to next option.

  . Current Hash algorithm for signature is SHA1.
  + Press SPACE to select Hash algorithm (SHA1/224/256/384/512) for signature, press ENTER to next option.

  . Current data length is 256 bytes.
  + Press SPACE to select data length (256 or 1024 or 4096), press ENTER to run.

  . Digital signature
  + ECC Weierstrass Prime - ECC P192
  + Generate a non-exportable wrapped asymmetric key...
Keyspec: 0x8900c417
  + Sign 256 bytes message with SHA1 and wrapped private key... SL_STATUS_OK (cycles: 125919 time: 3313 us)
  + Export public key from private key... SL_STATUS_OK (cycles: 115894 time: 3049 us)
  + Verify signature with SHA1 and wrapped public key... SL_STATUS_OK (cycles: 122195 time: 3215 us)

  . Current asymmetric key is a wrapped key.
  + Press SPACE to select a plaintext or wrapped or volatile key, press ENTER to next option.
```

### 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:




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".



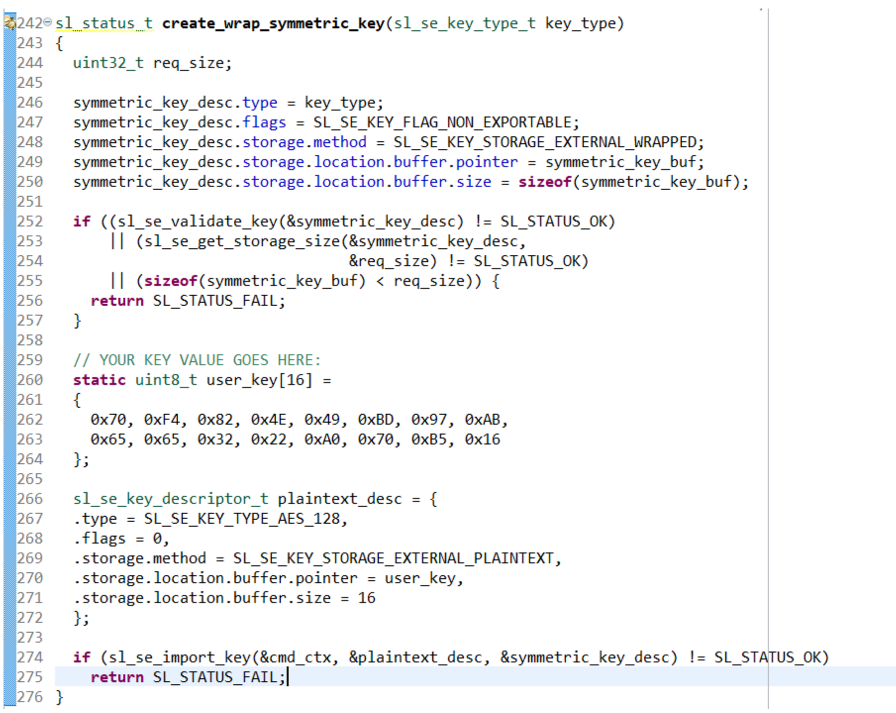
8. We will modify the "create_wrap_symmetric_key" function of app_se_manager_block_cipher.c to use our "CPMS key". Instead of generating a key, we will import our aes key. In **app_se_manager_block_cipher.c** line 259, replace the lines:

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

with the following:

```
// 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.



```
242 sl_status_t create_wrap_symmetric_key(sl_se_key_type_t key_type)
243 {
244   uint32_t req_size;
245
246   symmetric_key_desc.type = key_type;
247   symmetric_key_desc.flags = SL_SE_KEY_FLAG_NON_EXPORTABLE;
248   symmetric_key_desc.storage.method = SL_SE_KEY_STORAGE_EXTERNAL_WRAPPED;
249   symmetric_key_desc.storage.location.buffer.pointer = symmetric_key_buf;
250   symmetric_key_desc.storage.location.buffer.size = sizeof(symmetric_key_buf);
251
252   if ((sl_se_validate_key(&symmetric_key_desc) != SL_STATUS_OK)
253       || (sl_se_get_storage_size(&symmetric_key_desc,
254                                  &req_size) != SL_STATUS_OK)
255       || (sizeof(symmetric_key_buf) < req_size)) {
256     return SL_STATUS_FAIL;
257   }
258
259   // YOUR KEY VALUE GOES HERE:
260   static uint8_t user_key[16] =
261   {
262     0x70, 0xF4, 0x82, 0x4E, 0x49, 0xBD, 0x97, 0xAB,
263     0x65, 0x65, 0x32, 0x22, 0xA0, 0x70, 0xB5, 0x16
264   };
265
266   sl_se_key_descriptor_t plaintext_desc = {
267   .type = SL_SE_KEY_TYPE_AES_128,
268   .flags = 0,
269   .storage.method = SL_SE_KEY_STORAGE_EXTERNAL_PLAINTEXT,
270   .storage.location.buffer.pointer = user_key,
271   .storage.location.buffer.size = 16
272   };
273
274   if (sl_se_import_key(&cmd_ctx, &plaintext_desc, &symmetric_key_desc) != SL_STATUS_OK)
275     return SL_STATUS_FAIL;
276 }
```

9. Next, we need to write the wrapped key blob into flash. Add the following lines to **create_wrap_symmetric_key**:

```
// 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;
```

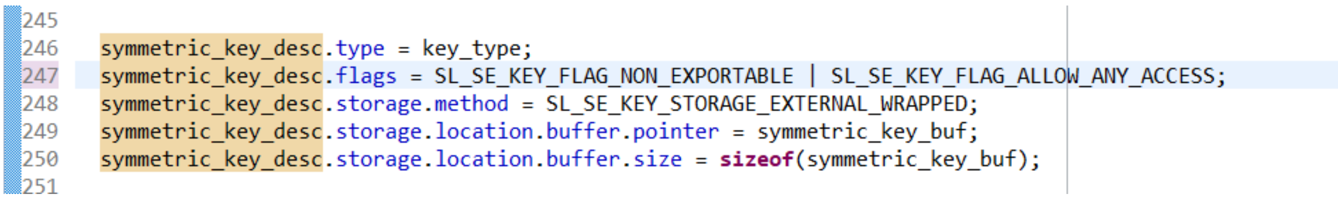
10. Next, we'll print out the keyspec that we need for CPMS. Add the following lines to **create_wrap_symmetric_key:**

```
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;
```

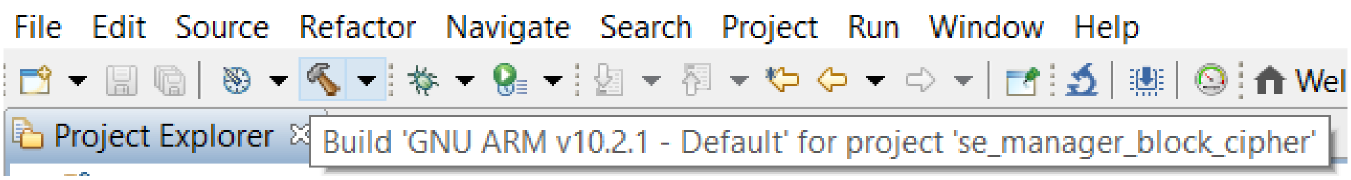
```
  if (sl_se_import_key(&cmd_ctx, &plaintext_desc, &symmetric_key_desc) != SL_STATUS_OK)
    return SL_STATUS_FAIL;

  // 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;

  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**):

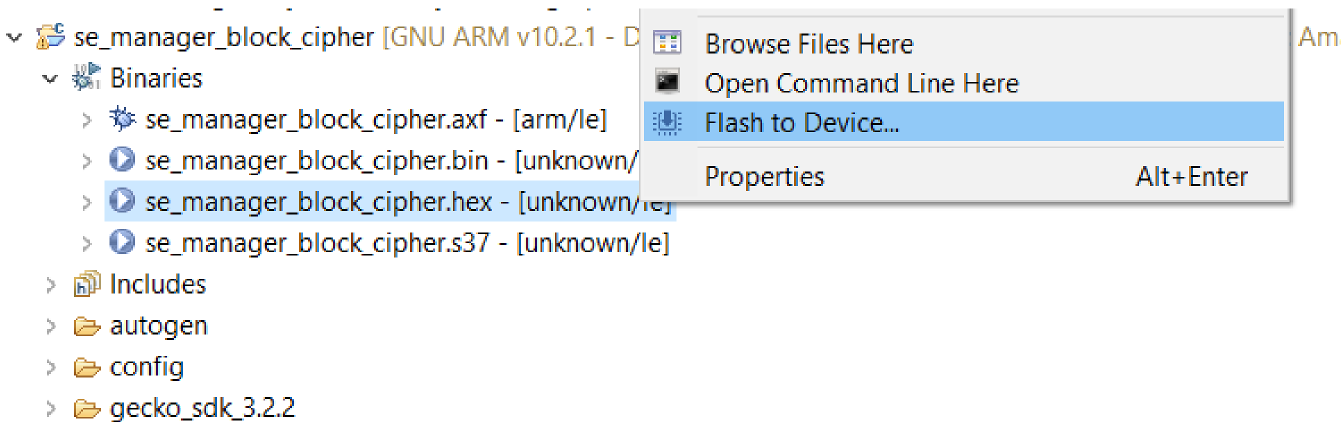
```
symmetric_key_desc.flags = SL_SE_KEY_FLAG_NON_EXPORTABLE | SL_SE_KEY_FLAG_ALLOW_ANY_ACCESS;
```



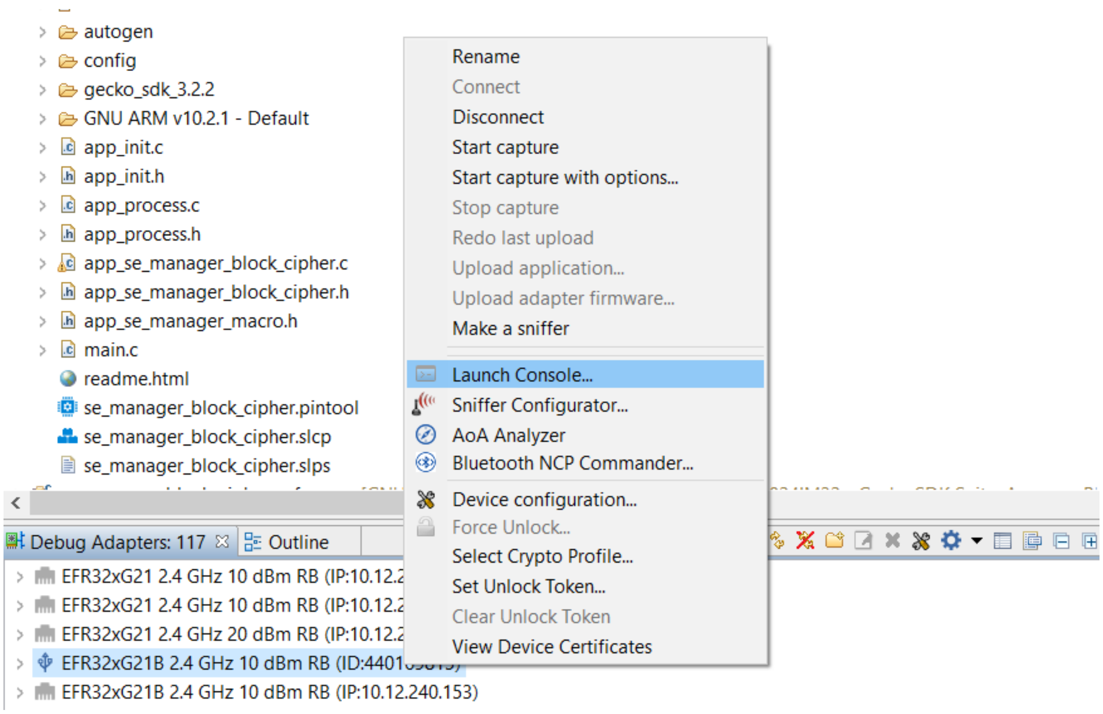

12. Build the project.




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".

```
Serial 0   Serial 1   Admin   Debug

SE Manager Block Cipher Example - Core running at 38000 kHz.
  . SE manager initialization... SL_STATUS_OK (cycles: 10 time: 0 us)

  . Fill buffers for block cipher operations.
  + Filling 16 bytes IV buffer with random number... SL_STATUS_OK (cycles: 3777 time: 99 us)
  + Filling 32 bytes associated data buffer with random number... SL_STATUS_OK (cycles: 3765 time: 99 us)
  + Filling 4096 bytes plain message buffer with random number... SL_STATUS_OK (cycles: 69629 time: 1832 us)

  . Current symmetric key is a plaintext key.
  + Press SPACE to select a plaintext or wrapped or volatile key, press ENTER to next option.
  + Current symmetric key is a wrapped key.

  . Current symmetric key length is 128-bit.
  + Press SPACE to select symmetric key length (128 or 192 or 256), press ENTER to next option.
```

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.

```
  . Current symmetric key length is 128-bit.
  + Press SPACE to select symmetric key length (128 or 192 or 256), press ENTER to next option.
  + Generating a 128-bit non-exportable symmetric wrapped key... Writing key into flash at 0x00080000...

Keyspec: 0x09008010

  . Current data length is 256 bytes.
  + Press SPACE to select data length (256 or 1024 or 4096), press ENTER to next option.
```

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.

```
Keyspec: 0x09008010

  . Current data length is 256 bytes.
  + Press SPACE to select data length (256 or 1024 or 4096), press ENTER to next option.

  . Current Hash algorithm for HMAC is SHA1.
  + Press SPACE to select Hash algorithm (SHA1/224/256/384/512) for HMAC, press ENTER to run.

  . AES ECB test
  + Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 15379 time: 404 us)
  + Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 15507 time: 408 us)
  + Comparing decrypted message with plain message... OK

  . AES CTR test
  + Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 15527 time: 408 us)
  + Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 15476 time: 407 us)
  + Comparing decrypted message with plain message... OK

  . AES CCM test
  + Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 16101 time: 423 us)
  + Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 17067 time: 449 us)
  + Comparing decrypted message with plain message... OK

  . AES GCM test
  + Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 16333 time: 429 us)
  + Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 16504 time: 434 us)
  + Comparing decrypted message with plain message... OK

  . AES CBC test
  + Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 15333 time: 403 us)
  + Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 15137 time: 398 us)
  + Comparing decrypted message with plain message... OK

  . AES CFB8 test
  + Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 3985747 time: 104 ms)
  + Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 3985039 time: 104 ms)
  + Comparing decrypted message with plain message... OK

  . AES CFB128 test
  + Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 15403 time: 405 us)
  + Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 15528 time: 408 us)
  + Comparing decrypted message with plain message... OK

  . AES CMAC test
  + Generating 16 bytes CMAC on 256 bytes message with 128 bit key... SL_STATUS_OK (cycles: 15491 time: 407 us)

  . HMAC test
  + Generating SHA1 HMAC on 256 bytes message with 128 bit key... SL_STATUS_OK (cycles: 14388 time: 378 us)
```

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.

```
257    }
258
259 //  // YOUR KEY VALUE GOES HERE:
260 //  static uint8_t user_key[16] =
261 //  {
262 //    0x70, 0xF4, 0x82, 0x4E, 0x49, 0xBD, 0x97, 0xAB,
263 //    0x65, 0x65, 0x32, 0x22, 0xA0, 0x70, 0xB5, 0x16
264 //  };
265 //
266 //  sl_se_key_descriptor_t plaintext_desc = {
267 //  .type = SL_SE_KEY_TYPE_AES_128,
268 //  .flags = 0,
269 //  .storage.method = SL_SE_KEY_STORAGE_EXTERNAL_PLAINTEXT,
270 //  .storage.location.buffer.pointer = user_key,
271 //  .storage.location.buffer.size = 16
272 //  };
273 //
274 //  if (sl_se_import_key(&cmd_ctx, &plaintext_desc, &symmetric_key_desc) != SL_STATUS_OK)
275 //    return SL_STATUS_FAIL;
276
277   // YOUR KEY ADDRESS GOES HERE:
278   unsigned int wrapped_key_address = 0x00080000;
279
280 //  printf("Writing key into flash at 0x%08x...\n",  wrapped_key_address);
281 //
282 //  // Clear out the old wrapped key
283 //  MSC_ErasePage((uint32_t*)wrapped_key_address);
284 //
285 //  // Flash the new wrapped key
286 //  MSC_WriteWord((uint32_t*)wrapped_key_address, symmetric_key_buf, sizeof(symmetric_key_buf));
287
288   // Update the key descriptor to point to the key in flash
289   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.

20. 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.

```
  . Current symmetric key is a plaintext key.
  + Press SPACE to select a plaintext or wrapped or volatile key, press ENTER to next option.
  + Current symmetric key is a wrapped key.

  . Current symmetric key length is 128-bit.
  + Press SPACE to select symmetric key length (128 or 192 or 256), press ENTER to next option.
  + Generating a 128-bit non-exportable symmetric wrapped key...
Keyspec: 0x09008010

  . Current data length is 256 bytes.
  + Press SPACE to select data length (256 or 1024 or 4096), press ENTER to next option.

  . Current Hash algorithm for HMAC is SHA1.
  + Press SPACE to select Hash algorithm (SHA1/224/256/384/512) for HMAC, press ENTER to run.

  . AES ECB test
  + Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 13905 time: 365 us)
  + Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 15018 time: 395 us)
  + Comparing decrypted message with plain message... OK

  . AES CTR test
  + Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 15474 time: 407 us)
  + Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 15532 time: 408 us)
  + Comparing decrypted message with plain message... OK

  . AES CCM test
  + Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 16497 time: 434 us)
  + Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 16856 time: 443 us)
  + Comparing decrypted message with plain message... OK

  . AES GCM test
  + Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 16159 time: 425 us)
  + Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 16464 time: 433 us)
  + Comparing decrypted message with plain message... OK

  . AES CBC test
  + Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 15469 time: 407 us)
  + Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 15106 time: 397 us)
  + Comparing decrypted message with plain message... OK

  . AES CFB8 test
  + Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 3985565 time: 104 ms)
  + Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 3978599 time: 104 ms)
  + Comparing decrypted message with plain message... OK

  . AES CFB128 test
  + Encrypting 256 bytes plaintext with 128 bit key... SL_STATUS_OK (cycles: 15176 time: 399 us)
  + Decrypting 256 bytes ciphertext with 128 bit key... SL_STATUS_OK (cycles: 15550 time: 409 us)
  + Comparing decrypted message with plain message... OK

  . AES CMAC test
  + Generating 16 bytes CMAC on 256 bytes message with 128 bit key... SL_STATUS_OK (cycles: 15370 time: 404 us)

  . HMAC test
  + Generating SHA1 HMAC on 256 bytes message with 128 bit key... SL_STATUS_OK (cycles: 14034 time: 369 us)
```

# 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**

**www.silabs.com**