PSoC® 64 Secure MCU

Secure Boot SDK User Guide

Document Number: 002-27860 Rev *B

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
http://www.cypress.com

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRA-NTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage (“Unintended Uses”). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Contents

1 Introduction

Cypress provides the Secure Boot SDK to simplify using the PSoC 64 Secure MCU line of devices. This SDK includes all required libraries, tools, and sample code to provision and develop applications for PSoC 64 MCUs.

The Secure Boot SDK provides provisioning scripts with sample keys and policies, a pre-built Cypress Secure Bootloader image, and post-build tools for signing firmware images.

Where to Get the Secure Boot SDK

The SDK is currently available as the CySecureTools library for Python and is in the Cypress Semiconductor GitHub site at: https://github.com/cypresssemiconductorco/cysecuretools.

Using this Guide

This guide provides a high-level overview of the Secure Boot SDK, including details on how the provisioning process works, as well as descriptions of the provided scripts and tools. In addition, this guide provides a reference of the tokens/JSON structures.

Definition of Terms

- **Root-of-Trust (RoT):** This is an immutable process or identity used as the first entity in a trust chain. No ancestor entity can provide a trustable attestation (in digest or other form) for the initial code and data state of the RoT.
- **Hardware Security Module (HSM):** A physical computing device that safeguards and manages digital keys for strong authentication, and that provides crypto processing. In the context of the PSoC 64 Secure MCU, the HSM is a device programming engine placed in a physically secure facility.
- **Provisioning:** The process by which keys, policies and secrets are injected into the device. Once provisioned, the device can be accessed or modified only with the keys injected adhering to the relevant policies injected.
- **JSON:** JavaScript Object Notation (JSON) is an open-standard file format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types (or any other serializable value).
- **JWT:** JSON Web Token (JWT) is an open, industry standard RFC 7519 method to securely represent claims between two parties.
- **JWK:** JSON Web Key (JWK) is a RF7517 compliant data structure that represents a cryptographic key.
- **Policies:** Policies are a collection of pre-defined (name,value) pairs that describe what is and is not allowed on the device to which the policy applies. Most policies are enforced by the hardware RoT firmware

in the device (implemented by boot code), and some are interpreted and enforced by higher layers of software like Cypress Secure Bootloader.

- **Secure Boot:** Refers to a bootup process where the firmware being run by the chip is trusted by utilizing strong cryptographic schemes and an immutable RoT.
- **KitProg3:** This is s PSoC 5LP device serving as KitProg3. It is a is a multi-functional system, which includes a SWD programmer, debugger, USB-I2C bridge, and USB-UART bridge.
- **SMIF:** Serial Memory interface. In the context of this user guide, it refers to the highspeed Quad-SPI interface on PSoC 6.

Revision History

Document Title: PSoC(R) 64 Secure MCU Secure Boot SDK User Guide
Document Number: 002-27860

Revision	Date	Description of Change
**	7/19/19	New document.
*A	9/18/19	Updates to change to CySecureTools flow.
*B	12/4/19	Updates to include ModusToolbox 2.0 flow.

2 Overview

The PSoC 64 Secure MCU line, based on the PSoC 6 MCU platform, features out-of-box security functionality. The line provides an isolated RoT with true attestation and provisioning services. In addition, these MCUs deliver a pre-configured secure execution environment which supports system software of various IoT platforms and provides:

- Secure provisioning
- Secure storage
- Secure firmware management

Developing with a PSoC 64 Secure MCU requires the chip to be provisioned with keys and policies, and then programmed with signed firmware for the device to boot up correctly. The Secure Boot SDK provides development tools to demonstrate the provisioning and signing flow. In addition, Cypress Secure Bootloader is included with this release.

Secure Boot SDK Components

The Secure Boot SDK is organized as a stand-alone python **CySecureTools** package, which contains all the required scripts, default provisioning packets, and the default policy file, as follows:

Component	Purpose
Provisioning Scripts	Python scripts for provisioning the PSoC 64 Secure MCU. Scripts are based on Python, OpenOCD and Kitprog3.
Entrance Exam Scripts	Runs an entrance exam on the PSoC 64 Secure MCU to ensure no tampering has occurred.
Cypress Secure Bootloader Image	The first stage bootloader based on an open source MCUBoot [1] library.
Sample Provisioning Policies	Examples to be used as templates for forming provisioning tokens.

What is Provisioning?

Provisioning is a process whereby secure assets like keys and security policies are injected into the device. This step typically occurs in a secure manufacturing environment that has a Hardware Security Module (HSM). For the PSoC64 Secure MCU, provisioning involves the following steps:

- Transferring the RoT from Cypress to the development user (called OEM in this document).
- Injecting user assets such as image-signing keys, device security policies, and certificates into the device.

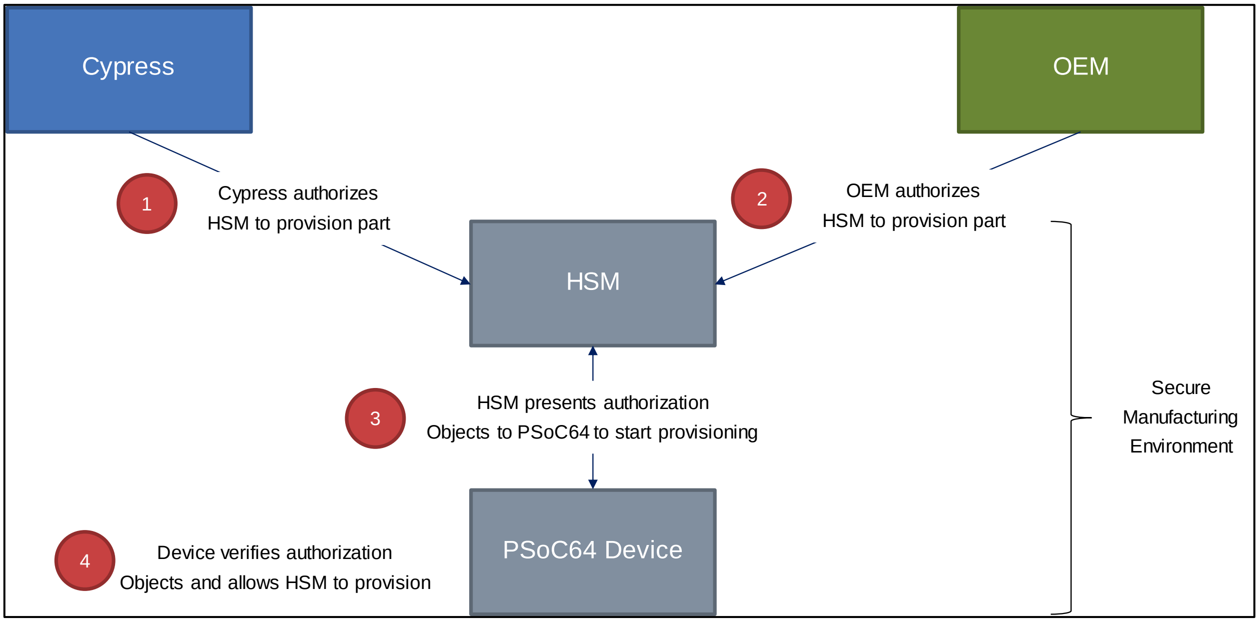
[1] https://mcuboot.com/

Transferring RoT

Every PSoC 64 Secure MCU has a Cypress public key present in the part during manufacturing. This Cypress public key acts as the RoT for the device once it is manufactured.

The RoT transfer process can be represented as a series of trust claims; exchanged between the following entities:

- Cypress – The owner of the Cypress Root private key.
- Secure Manufacturing environment HSM – The entity authorized to provision and program the PSoC 64 Secure MCU.
- OEM/Developer – The user/code developer of the part.
- PSoC 64 Secure MCU – The holder of the Cypress Root public key.


Cypress
OEM
1
Cypress authorizes HSM to provision part
2
OEM authorizes HSM to provision part
HSM
3
HSM presents authorization Objects to PSoC64 to start provisioning
Secure Manufacturing Environment
4
Device verifies authorization Objects and allows HSM to provision
PSoC64 Device

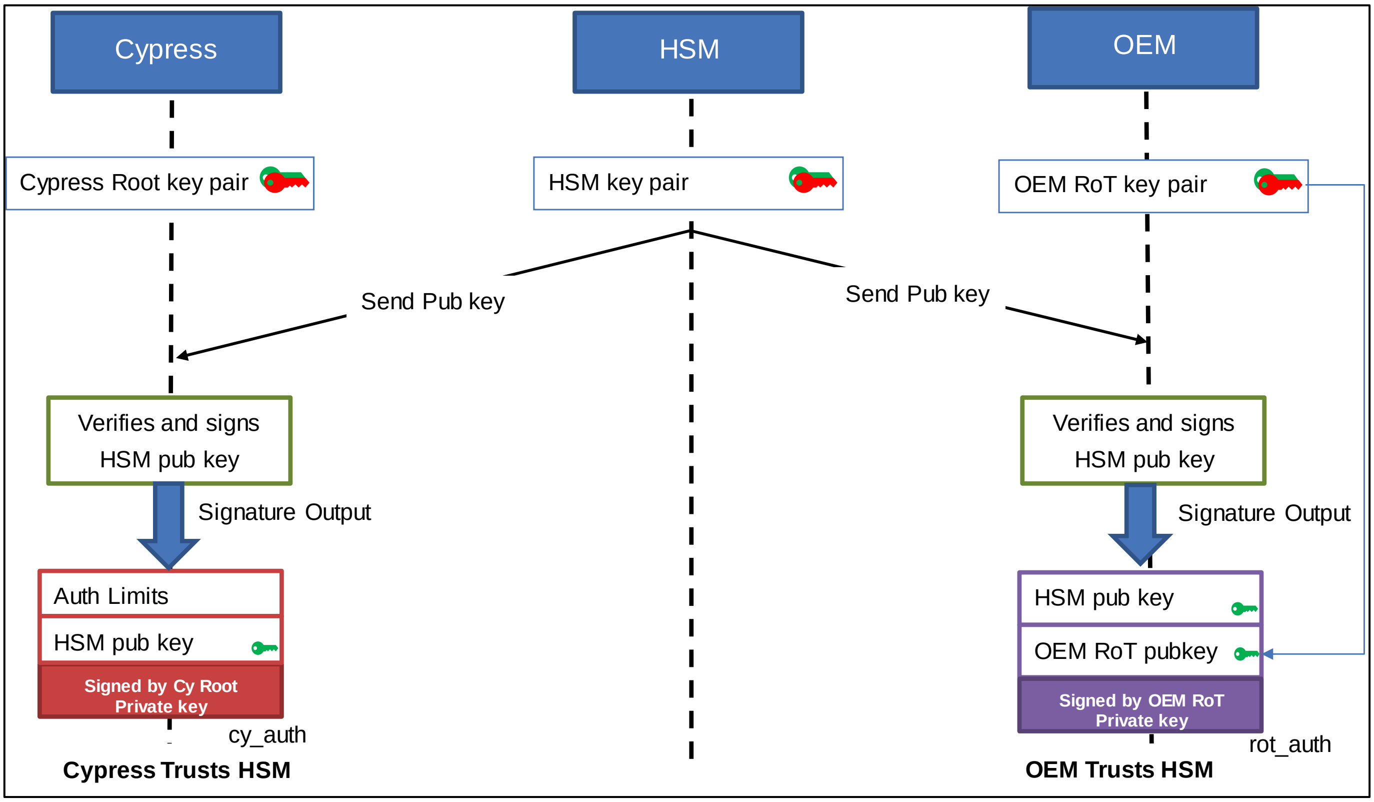
The following steps reflect the corresponding numbers in the diagram:

1. Cypress authorizes the HSM to provision a part.
2. The OEM/User authorizes the same HSM to provision the part with credentials and firmware.
3. The HSM then presents the above authorization objects to the PSoC 64 Secure MCU.
4. The PSoC 64 Secure MCU verifies authorization signatures and claims. If all are valid, the chip accepts the OEM RoT public key and allows the HSM to further send provisioning packets.

The end result of this RoT transfer process can be represented as follows:

- The PSoC 64 Secure MCU now uses the OEM RoT public key as the root key used to validate any OEM asset (image keys, policies etc.).
- The PSoC 64 Secure MCU now trusts the HSM public key and expects further provisioning packets to be signed by the corresponding HSM private key.

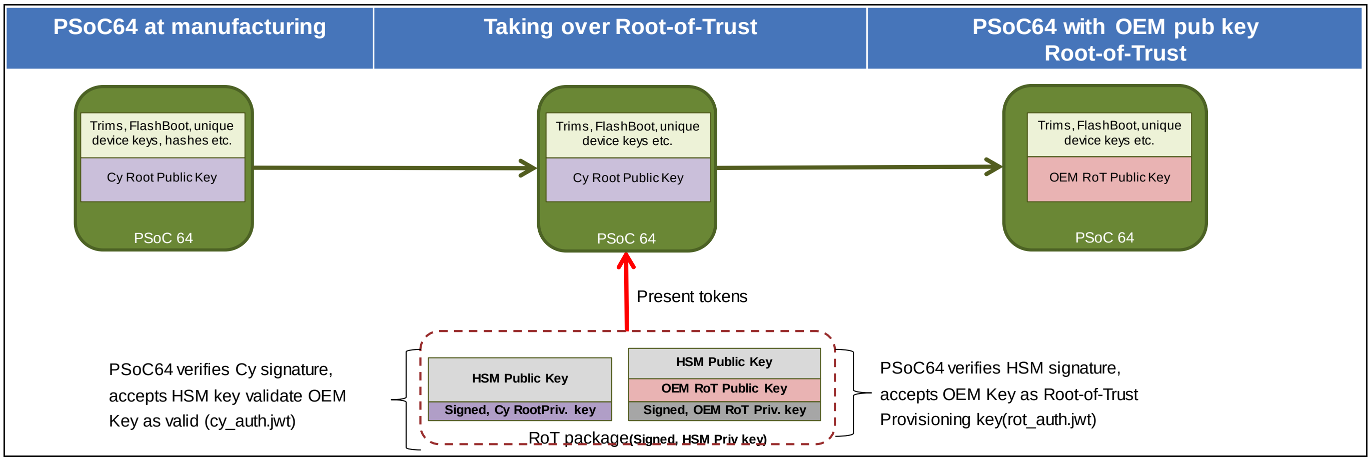
The actual authorization objects for the PSoC 64 Secure MCU are represented using the JSON Web Token (JWT) format. The authorization flow of the Cypress and the User authorizing a HSM is shown in the following diagram:



The final output of this process generates the following JWTs:

- **cy_auth JWT:** Contains the public key of the HSM to be trusted. Additional fields such as an expiration date can be specified to limit this token's usage.
- **rot_auth JWT:** Contains the public key of the HSM to be trusted as well as the OEM RoT public key to which the RoT must be transferred.

The HSM then presents these tokens to the chip, as shown in the following diagram:

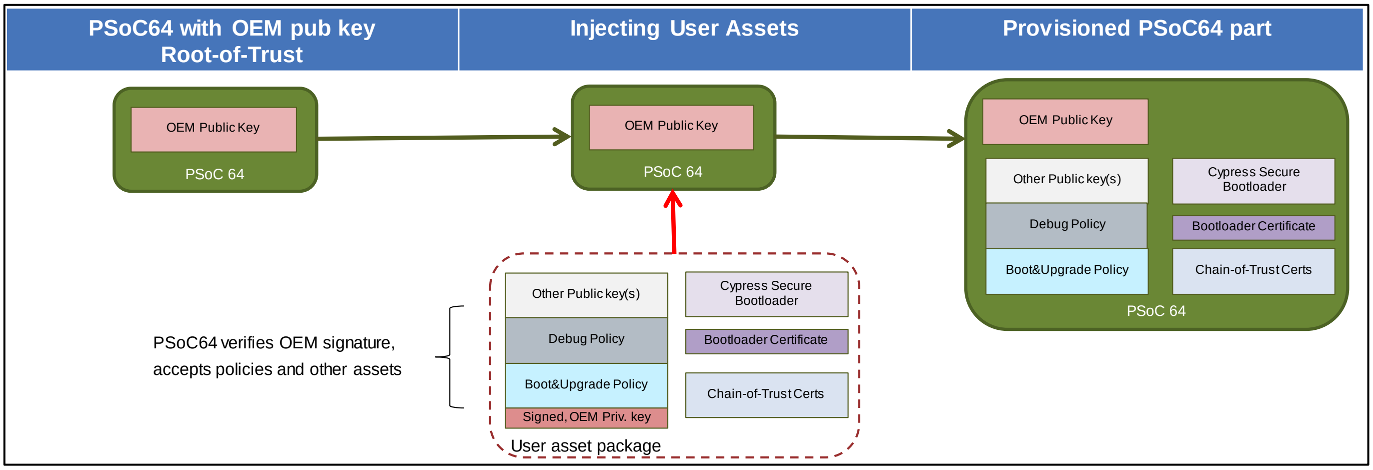

PSoC64 at manufacturing
Taking over Root-of-Trust
PSoC64 with OEM pub key Root-of-Trust
Trims, FlashBoot, unique device keys, hashes etc.
Cy Root Public Key
PSoC 64
Trims, FlashBoot, unique device keys etc.
Cy Root Public Key
PSoC 64
Trims, FlashBoot, unique device keys etc.
OEM RoT Public Key
PSoC 64
Present tokens
PSoC64 verifies Cy signature, accepts HSM key validate OEM Key as valid (cy_auth.jwt)
HSM Public Key
Signed, Cy RootPriv. key
HSM Public Key
OEM RoT Public Key
Signed, OEM RoT Priv. key
RoT package(Signed, HSM Priv key)
PSoC64 verifies HSM signature, accepts OEM Key as Root-of-Trust Provisioning key(rot_auth.jwt)

Injecting User Assets

Once the RoT is transferred to the OEM RoT public key, the user can inject several assets into the device. They are primarily divided into the following types of assets:

- Public Keys
  - Image public key – Used by the Bootloader to check the next image signature.
- Device Policies
  - Boot & Upgrade policy – Specifies which regions of flash constitute a bootloader and launch image, as well as the key associated when validating the flash area.
  - Debug policy – Specifies the behavior of the device debug ports (CM0+/CM4/SYSAP). Also, specifies the device behavior when transitioning into RMA mode.
- Chain-of-Trust Certificates
  - Any certificates that need to be injected into devices; for example, device certificate for TLS or Identity.

Both public keys and device policies are present in a JWT token called ‘prov_req.JWT.’ They are signed by the OEM RoT private key. The certificates present in the chain-of-trust can be signed by the same key, but no restrictions are placed on this field’s contents as the chain-of-trust can roll up to a higher authority.



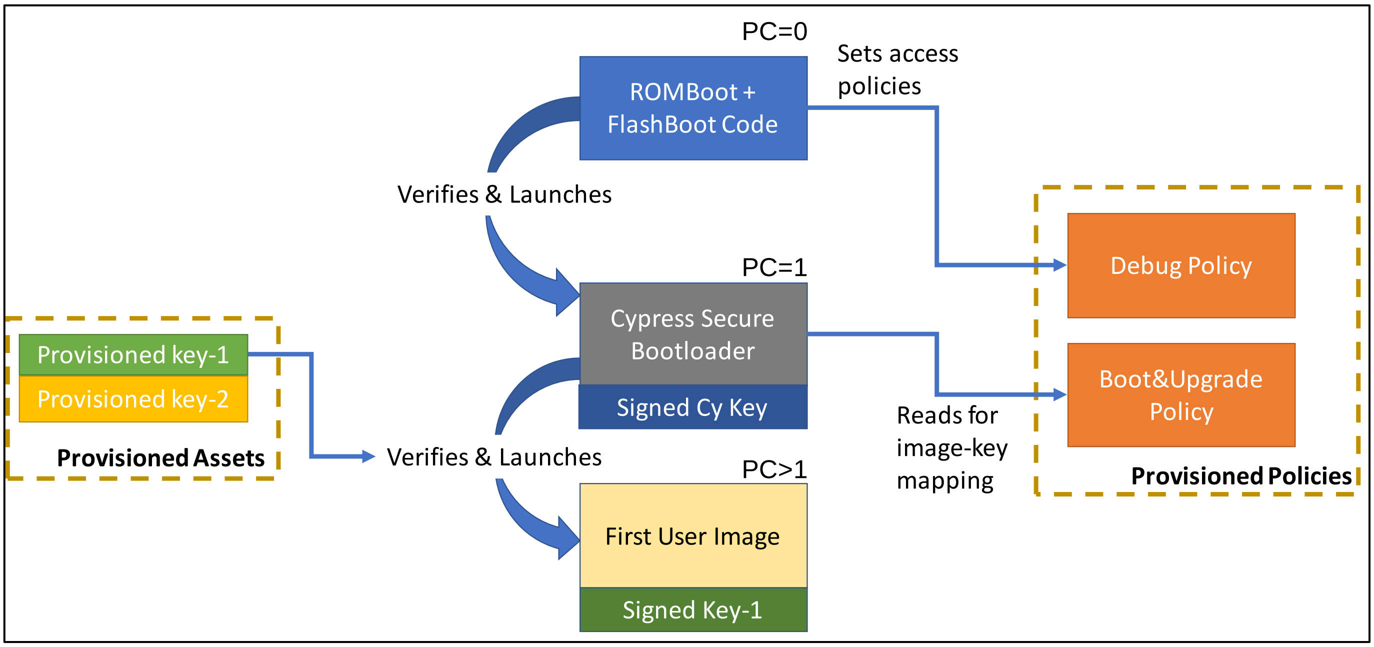
In addition to the OEM assets, Cypress Secure Bootloader also gets programmed at this stage, along with 'image_cert.JWT’ that has the signature of the Cypress Secure Bootloader binary. For more details on Cypress Secure Bootloader, see the Cypress Secure Bootloader section.

For more details on the exact provisioning packets, see the Provisioning script flow details section.

Cypress Secure Bootloader

The Cypress Secure Bootloader is included as a pre-built hex image. This image acts as the first image securely launched by the PSoC 64 Secure MCU boot code. The Cypress Secure Bootloader is a port of the open source MCUBoot library and is capable of parsing the provisioned Boot&Upgrade policy to launch a next image. For more details about this open source library, refer to the MCUBoot Bootloader design website.

The Cypress Secure Bootloader also enforces the protection contexts for the bootloader code, so code running on another protection context cannot overwrite/tamper with the boot code. The following diagram shows the launch code sequence of Cypress Secure Bootloader:



During a normal bootup, the Cypress Secure Bootloader performs the following operations:

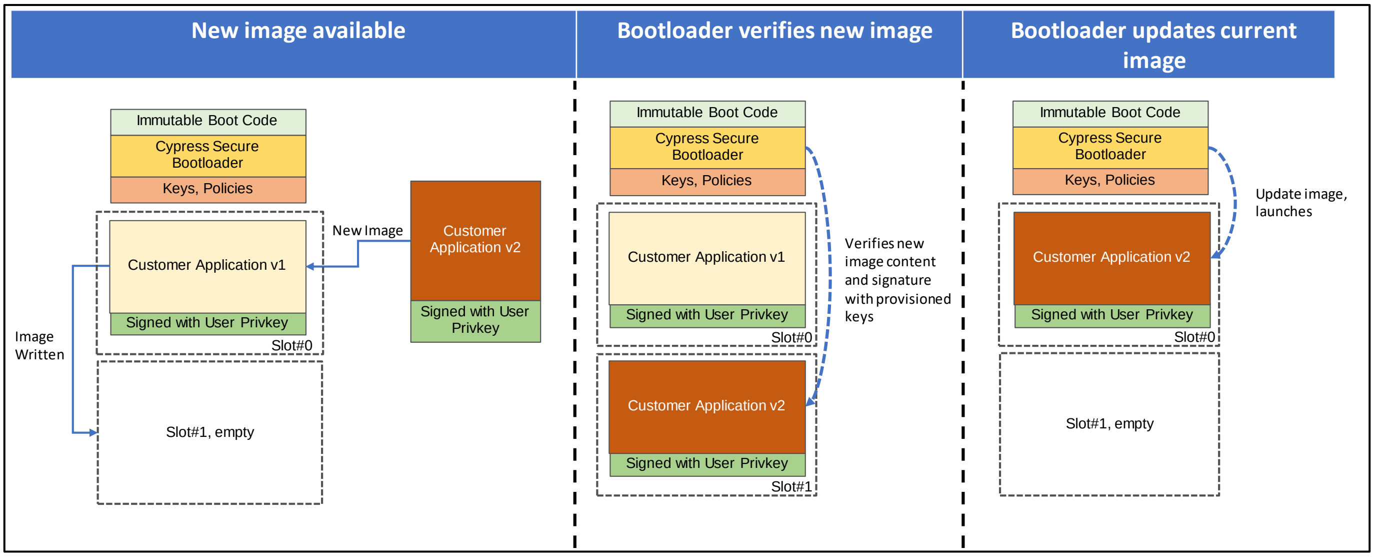
- Reads the policies and parses them for further use
- Checks if Boot Area (Slot 0) contains an image to boot
- Verifies if this image has the valid format
- Verifies the image's digital signature
- Verifies if the image rollback counter is bigger or equal to the one saved in the rollback protection counter.

If an upgrade image exists, it performs the following operations:

- Checks if Staging Area (Slot 1) has an image for upgrade
- Boots Slot 0 if no correct image found in Staging Area
- If Slot 1 has a new image, verifies its digital signature
- Decrypts the image’s body if the signature is valid (optional for the encrypted image support)
- Verifies the digital signature of the decrypted image (optional for the encrypted image support)

- Checks if the image metadata matches the image in Slot 0
- Checks if corresponding policies allow upgrading
- Rewrites Slot_0 with the decrypted (if needed) Slot_1 image
- Invalidates Slot_1 by erasing the header and trailer (hash and signature) sections, so that the next reset Slot_1 is ignored.

The following diagram shows a typical application update scenario using the Cypress Secure Bootloader:



Note 1 By default, the pre-built Cypress Secure Bootloader launches the CM4 core.

Note 2 The Cypress Secure Bootloader signature is checked only once during the provisioning operation (injection of user's assets). If the signature is valid, the Bootloader contents are HASH'ed along with other assets and blown in eFUSE. The boot code will check for a HASH match after that point and does not check for the signature again.

CySecureTools Installation and Documentation

The source code for CySecure tools can be found on git and full details on the methods available can be found on https://github.com/cypresssemiconductorco/cysecuretools/blob/master/README.md/

The Secure Boot SDK is organized as a stand-alone python **CySecureTools** package, which contains all the needed sets of scripts, default provisioning packets, and a set of default policy files.

Python is required to be installed on your computer.

1. Install Python 3.7.0 or later on your computer. You can download it from https://www.python.org/downloads/.
2. Add the *python.exe* file location to the system variable "Path." For example: *C:\Python37\python.exe*.
3. Add the Python home folder/Scripts subfolder to the system variable "Path." For example: *C:\Python37\Scripts*.

4. Set up the appropriate environment variable:

   - **Windows:** If Python 2.7 installed also installed in the computer, make sure that Python37 and Python37\Scripts have higher priority in PATH than C:\Python27

     a. Open Control Panel, go to **System > Advanced System Settings > Environment Variables**.

     b. Find “PATH” in the list of user variables.

     c. Click **Edit**.

     d. Move “C:\Python37” and “C:\Python37\Scripts\” to the top.

   - **Linux:** Most distributions of Linux should already have python2 and python3 installed. To verify that python by default points to python3 run:

     ```
     python –version
     ```

     If python3 is not set as default run following commands. The number at the end of each command denotes a priority

     ```
     update-alternatives --install /usr/bin/python python /usr/bin/python2.7 1
     update-alternatives --install /usr/bin/python python /usr/bin/python3.7 2
     ```

   - **MacOS:** By default, 'python' points to /usr/bin/python which is Python2. To make 'python' and 'pip' resolve to Python3 versions, execute the following:

     ```
     echo 'alias python=python3' >> ~/.bash_profile
     echo 'alias pip=pip3' >> ~/.bash_profile
     source ~/.bash_profile
     python --version
     Python 3.7.4
     pip --version
     pip 19.0.3 from /Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
     packages/pip (python 3.7)
     ```

5. Install Secure Boot SDK package. Run the following command in your terminal window:

   ```
   pip install git+https://github.com/cypresssemiconductorco/cysecuretools.git
   ```

   Or simply:

   ```
   pip install cysecuretools
   ```

Note: During installation, there can be possible errors when installing colorama, protobuf and jsonschema. These can be safely ignored. For reference, you can use the following command to show the path to the installed package (and to the default policy file):

```
pip show cysecuretools
```

3 Mbed OS – Provisioning Flow

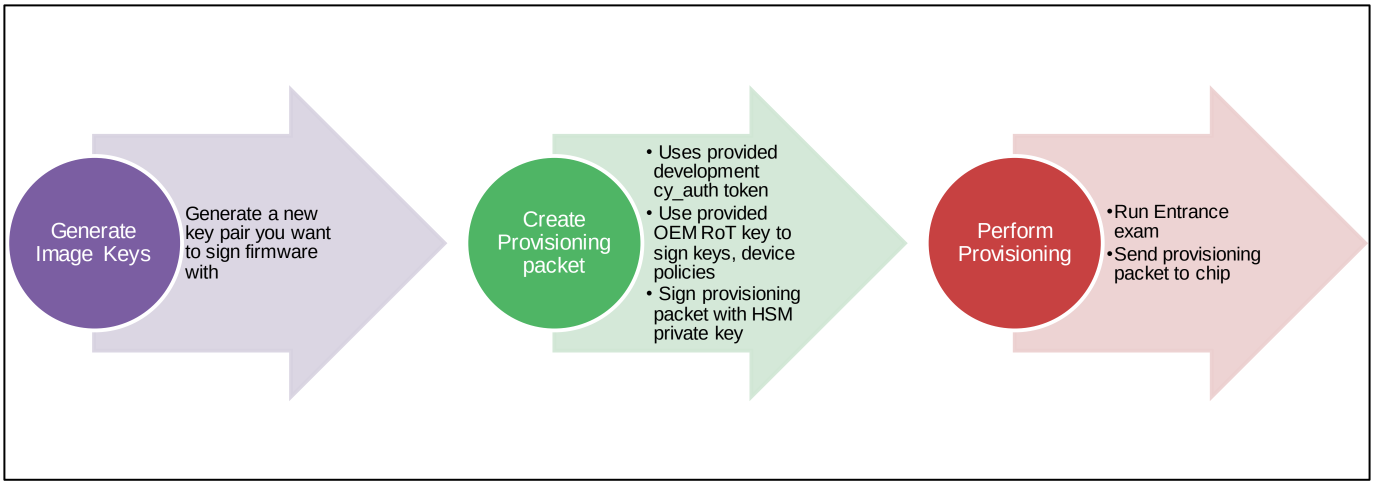
This section shows how to provision the CY8CPROTO-064S1-SB kit in Mbed OS using CySecureTools.

Prerequisites

- CySecureTools: Refer to the instructions in CySecureTools Installation and Documentation.
- Mbed Installation: Install Mbed CLI tools using the instructions at the Mbed OS website.

Device Provisioning

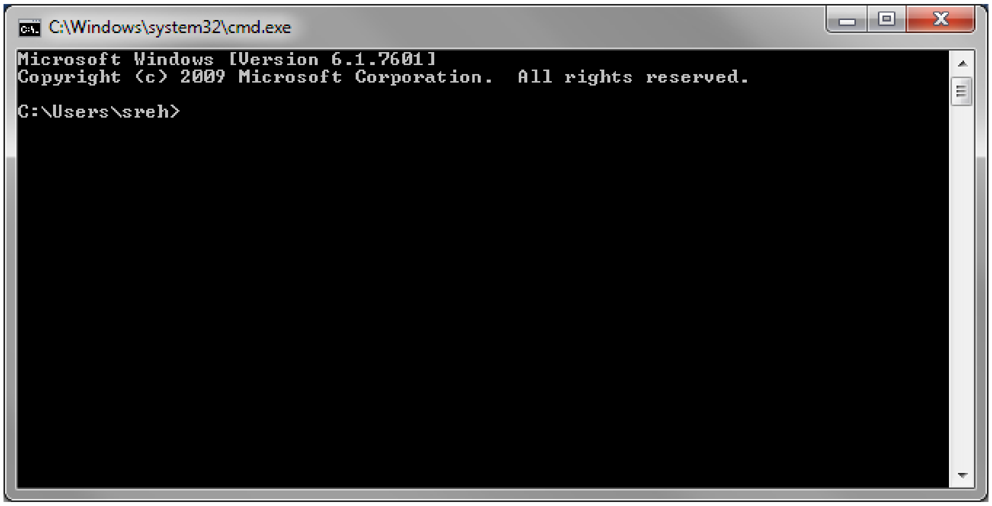
For evaluation, the device provisioning flow can be done on your local development environment. For evaluation, a pre-signed development token is available in the SDK which authorizes a HSM key-pair provided in the SDK.


Generate Image Keys
Generate a new key pair you want to sign firmware with
Create Provisioning packet
• Uses provided development cy_auth token
• Use provided OEM RoT key to sign keys, device policies
• Sign provisioning packet with HSM private key
Perform Provisioning
•Run Entrance exam
•Send provisioning packet to chip

Execute all the following steps from a command line. The path to the policy file can be relative to the current working directory or absolute. All paths to key files inside the policy file are relative to the path mentioned in the policy. For a detailed description of what the default policy file looks like, refer to the Understanding the Default policy section.

A. Setup mbed-os-example-blinky project folder:

1. Open your native command-line application, for example on Windows7



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\sreh>
```

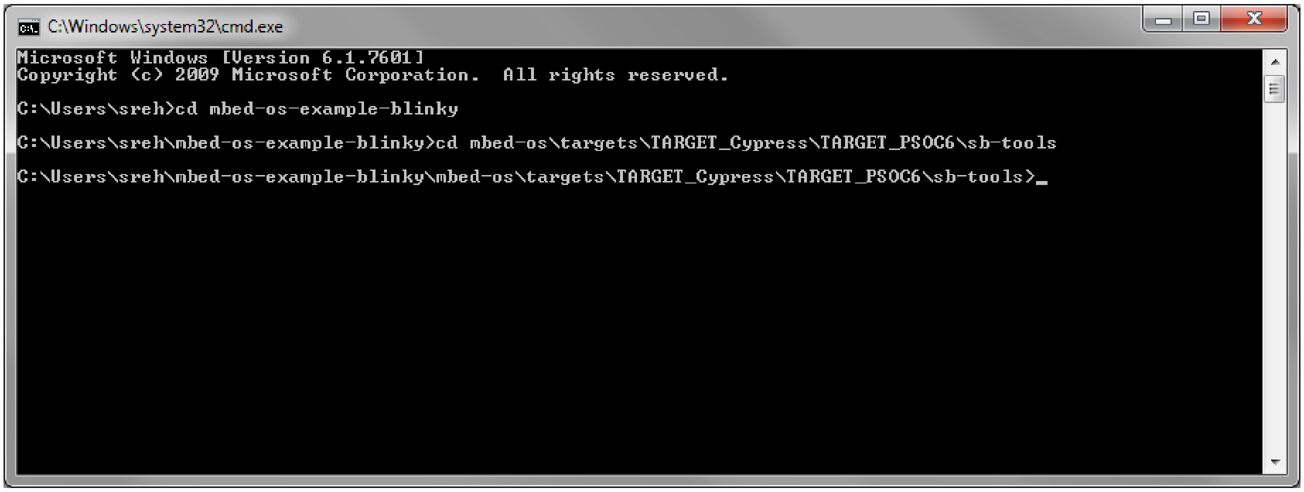
2. Import the blinky example project using the following command:

```
    mbed import mbed-os-example-blinky
```

3. Update to the latest version of Mbed OS using the following commands:

```
    cd mbed-os-example-blinky
    mbed update latest
```

4. Navigate to the *mbed-os\targets\TARGET_Cypress\TARGET_PSOC6\sb-tools\* directory. For example on Windows7,



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\sreh>cd mbed-os-example-blinky

C:\Users\sreh\mbed-os-example-blinky>cd mbed-os\targets\TARGET_Cypress\TARGET_PSOC6\sb-tools

C:\Users\sreh\mbed-os-example-blinky\mbed-os\targets\TARGET_Cypress\TARGET_PSOC6\sb-tools>_
```

Run the following command:

```
*Copy default policy from cysecuretools into mbed target*

*Example usage for the CY8CPROTO-064S1-SB kit*

python -c "import os; import shutil; import cysecuretools; os.makedirs('policy') if not
os.path.exists('policy') else None;
shutil.copy2(os.path.join(os.path.dirname(cysecuretools.__file__),
'targets/cy8cproto_064s1_sb/policy/policy_single_stage_CM4.json'),
'./policy/policy_single_stage_CM4.json')"
```

Note As an alternative, you can manually replace the policy file from your local python CySecureTools installation into the *sb-tools\policy* folder.

To use external memory (via SMIF) as staging (upgrade) area (slot_1) of the CM4 image, use *policy_single_stage_CM4_smif.JSON*.

B. Generate new keys:

Note All key descriptions in the policy file should be done before calling key generation. The key generation function generates all keys from the policy file, including the encryption key.

Ensure you are in the *mbed-os\targets\TARGET_Cypress\TARGET_PSOC6\sb-tools\* directory.

```
*Create keys, using specified policy:*

*Example usage for the CY8CPROTO-064S1-SB kit*

python -c "from cysecuretools import CySecureTools;tools = CySecureTools('CY8CPROTO-
064S1-SB', 'policy/policy_single_stage_CM4.json'); tools.create_keys();"

*Generic usage example*

python -c "from cysecuretools import CySecureTools;tools = CySecureTools('<TARGET>',
'policy/policy_single_stage_CM4.json'); tools.create_keys();"
```

<TARGET> will be the kit name, for example: CY8CPROTO-064S1-SB.

Additional parameters for `create_keys` are `overwrite` and `out`. The parameter `overwrite` specifies behavior when `create_key` finds keys in the output path. The value `None` (is value by default) means that you will be asked about overwriting such keys or not. `True` means overwrite keys, `False` means keep existing. The parameter `out` specifies the output directory for keys and overwrites key paths from the policy file.

C. Create provisioning packets:

```
*Create provisioning packets, using specified policy:*

*Example usage for the CY8CPROTO-064S1-SB kit*

python -c "from cysecuretools import CySecureTools; tools = CySecureTools('CY8CPROTO-
064S1-SB', 'policy/policy_single_stage_CM4.json'); tools.create_provisioning_packet();"

*Generic usage example*

python -c "from cysecuretools import CySecureTools; tools = CySecureTools('<TARGET>',
'policy/policy_single_stage_CM4.json'); tools.create_provisioning_packet();"
```

<TARGET> will be the kit name. for example CY8CPROTO-064S1-SB.

Note It is possible to choose between two variants of CyBootloader to be used during provisioning process – with or without diagnostic logs. For this, change the **cy_bootloader** section of the policy file:

```
CyBootloader without diagnostic logs
"cy_bootloader":
  {
    "mode": "release"
  }
CyBootloader with diagnostic logs
"cy_bootloader":
  {
    "mode": "debug"
  }
```

D. Run entrance exam:

Connect the kit to your PC.

ATTENTION The KitProg3 must be in CMSIS-DAP mode for provisioning. Press the ‘Mode’ button on the kit until the Status LED blinks slowly. For more details, refer to the KitProg3 User Guide.

```
*Run entrance exam *

*Example usage for the CY8CPROTO-064S1-SB kit*

python -c "from cysecuretools import CySecureTools; tools = CySecureTools('CY8CPROTO-
064S1-SB', 'policy/policy_single_stage_CM4.json'); tools.entrance_exam();"


*Generic usage example*

python -c "from cysecuretools import CySecureTools; tools = CySecureTools('<TARGET>',
'policy/policy_single_stage_CM4.json'); tools.entrance_exam();"
```

<TARGET> will be the kit name, for example: CY8CPROTO-064S1-SB.

The Entrance exam is a test routine that does the following things:

- Verify if Device is in SECURE UNCLAIMED mode

- Verify if FlashBoot has not been modified/tampered
- Verify if User flash is empty and no code is running before any provisioning takes place

Failing the entrance exam returns an error in the command line. If there is any firmware running on the device, it can be erased using tools like the Cypress Programmer. The mbed-cli does not natively provide any commands to erase the chip.

Note that the entrance exam is also automatically run before performing provisioning, so you can skip this step.

E. Perform provisioning:

ATTENTION The PSoC 6 supply voltage of 2.5 V is required to perform provisioning. Refer to the relevant kit user guide to find out how to change the supply voltage of your kit.

Execute the following command for device provisioning:

```
*Provision chip, using specified policy:*

*Example usage for the CY8CPROTO-064S1-SB kit*

python -c "from cysecuretools import CySecureTools;tools = CySecureTools('CY8CPROTO-
064S1-SB', 'policy/policy_single_stage_CM4.json'); tools.provision_device();"

*Generic usage example*

python -c "from cysecuretools import CySecureTools;tools = CySecureTools('<TARGET>',
'policy/policy_single_stage_CM4.json'); tools.provision_device();"
```

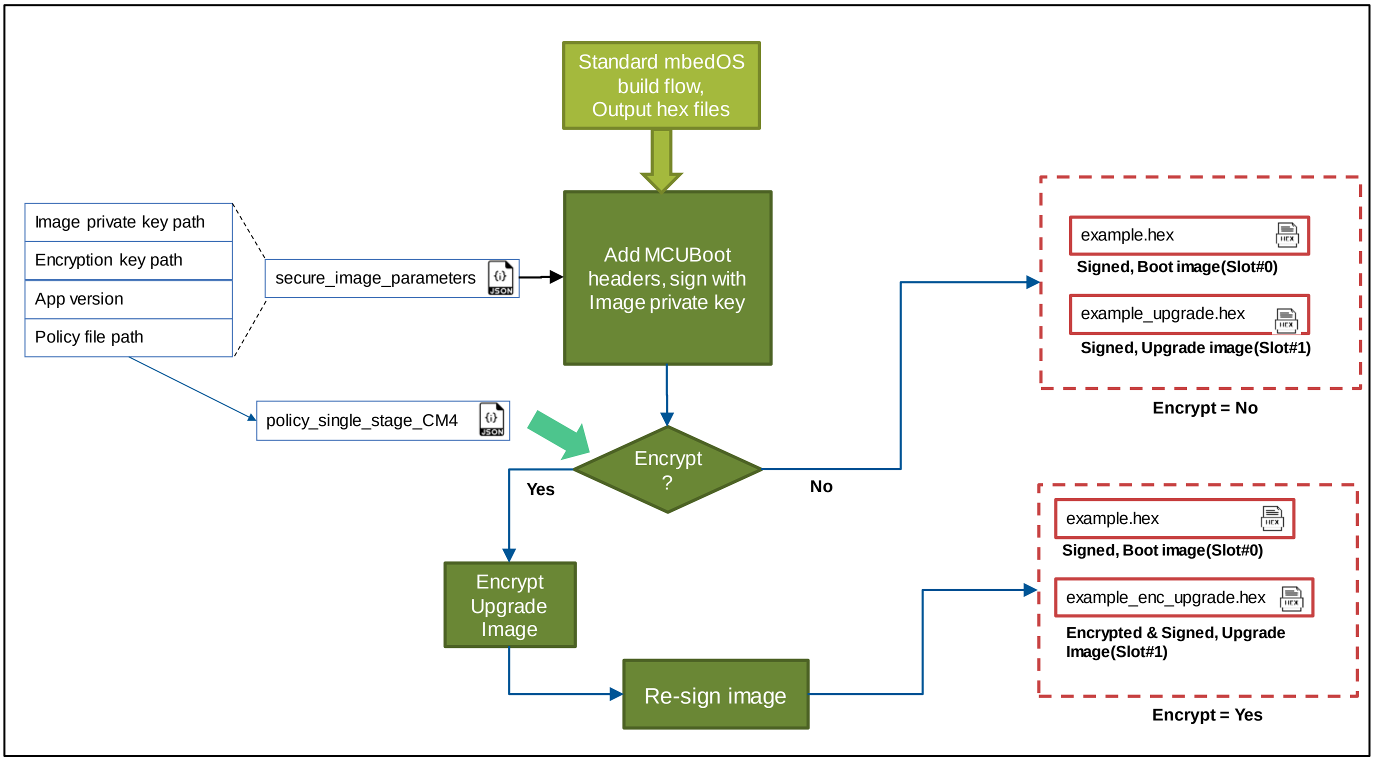
<TARGET> will be the kit name, for example: CY8CPROTO-064S1-SB.

Mbed OS Secure Image Generation

In Mbed OS, PSoC 64 based kit targets have post-build signing scripts set up so the output binary is formatted and signed automatically according to:

- The provisioned policy file; for example, policy_single_stage_CM4.json.
- The secure_image_parameters.json file in the target folder; for example, in *\TARGET_PSOC6\TARGET_CY8CPROTO_064_SB\secure_image_parameters.json*.

The following diagram shows the flow for signing and encryption using Mbed OS.



The build process outputs two binaries:

- Signed boot image hex file. This is the exact binary which can be programmed to Slot#0 for the PSoC 64 to securely launch the application.
- Signed and encrypted (optional) update image hex file. This is the exact binary that can be programmed to Slot#1 for the PSoC 64 to perform a secure update and then launch the application.

The following are descriptions of the *secure_image_parameters.json* fields:

- "boot0/VERSION": Specifies the version number of the Slot#0 image which will be placed in the image header
- "boot0/ROLLBACK_COUNTER": Specifies the counter value of the Slot#0 image
- "boot1/VERSION": Specifies the version number of the Slot#1 image which will be placed in the image header

- "boot1/ROLLBACK_COUNTER”: Specifies the counter value of the Slot#1 image
- "sdk_path": Specifies relative path to the sb-tools folder in mbed
- “policy_file”: Specifies relative path to the policy provisioned in the PSoC64
- “priv_key_file”: Specifies relative path to the private key used to sign the built application
- "aes_key_file": Specifies relative path to the private key used to encrypt the built application; note that this parameter is only used if “Encrypt” is set as “true” in the boot_upgrade field for the M4 image(“boot_auth”:8)
- "dev_pub_key_file": Specifies relative path to the device public key; note that this parameter is only used if “Encrypt” is set as “true” in the boot_upgrade field for the M4 image(“boot_auth”:8)

Encrypting Generic Images

A generic HEX file (for example one that is produced by Mbed OS or any other build system) can also be converted into a signed/encrypted image by executing the following command:

```
*Sign (and encrypt) image:*

python -c "from cysecuretools import CySecureTools;tools = CySecureTools('<TARGET>', 
'policy/policy_single_stage_CM4.json'); tools.sign_image('path/to_hex_file.hex');"
```

Building the Mbed OS Blinky Application

1. Build the application using the following command:

```
*Example usage for the CY8CPROTO-064S1-SB kit*
mbed compile -m CY8CPROTO_064_SB -t GCC_ARM
```

2. Program the built hex file located as follows:

   <Example Project>\BUILD\CY8CPROTO_064_SB\<project name>.hex

   Or run this command:

```
mbed compile -m CY8CPROTO_064_SB -t GCC_ARM -f
```

3. Reset the chip and observe blinking LED.

Debugging Application Code

Currently, Mbed OS export to ModusToolbox IDE is not fully supported for PSoC 64 based kits. This section describes how to use the compiled ELF file and PyOCD GDB server to debug the code.

Prerequisites

The following are required in order to debug application code:

- Application image is compiled and programmed into the device. The **mbed-os-example-blinky** is used as an example (https://github.com/ARMmbed/mbed-os-example-blinky).
- ModusToolbox IDE Version 2.0 or higher is installed on your machine.
- libusb driver is installed.
  - For **Windows**:
    - Download and unzip libusb-1.0.21.7z from https://github.com/libusb/libusb/releases/tag/v1.0.21.
    - Copy libusb-1.0.dll file into Python 3.7 folder (use 64-bit version of the DLL for 64-bit Python and 32-bit version of the DLL for 32-bit Python).
    - Make sure Python path located at the beginning of Path environment variable.
  - For **Linux / mac OS**: Use package manager to install libusb.

Create Empty C/C++ Application

1. Open the ModusToolbox IDE.
2. Go to **File->New->Project**.
3. Under **C/C++** select **C/C++ Project** and press **Next**.
4. Under **New C/C++ Project** select **C Managed Build** and press **Next**.
5. Fill in **Project name** and press **Next**, then **Next**, and **Finish**.

Configure PyOCD GDB Server Path

On the **Window** menu item, choose **Preferences > MCU > Global pyOCD Path** and set:

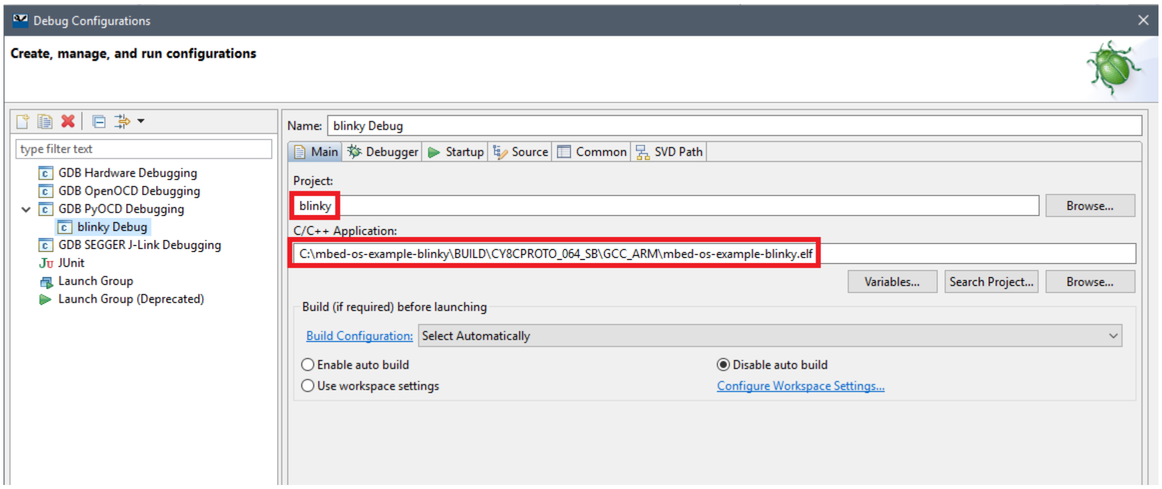
- **Executable**: pyocd-gdbserver
- **Folder**: C:\Python37-32\Scripts (align the path to Scripts directory in your python installation)

Setting up PyOCD Configuration for Debugging

1. Open **Run > Debug Configurations**.
2. Right click on GDB PyOCD Debugging and select **New**.
3. Configure the **Main** tab:
   a. Set Project to **blinky**.

b. Ensure the C/C++ Application points to the .elf file of the application.

c. Select **Disable auto build** check box to prevent the app building in the IDE.



Create, manage, and run configurations

Name: blinky Debug

Project: blinky

C/C++ Application: C:\mbed-os-example-blinky\BUILD\CY8CPROTO_064_SB\GCC_ARM\mbed-os-example-blinky.elf

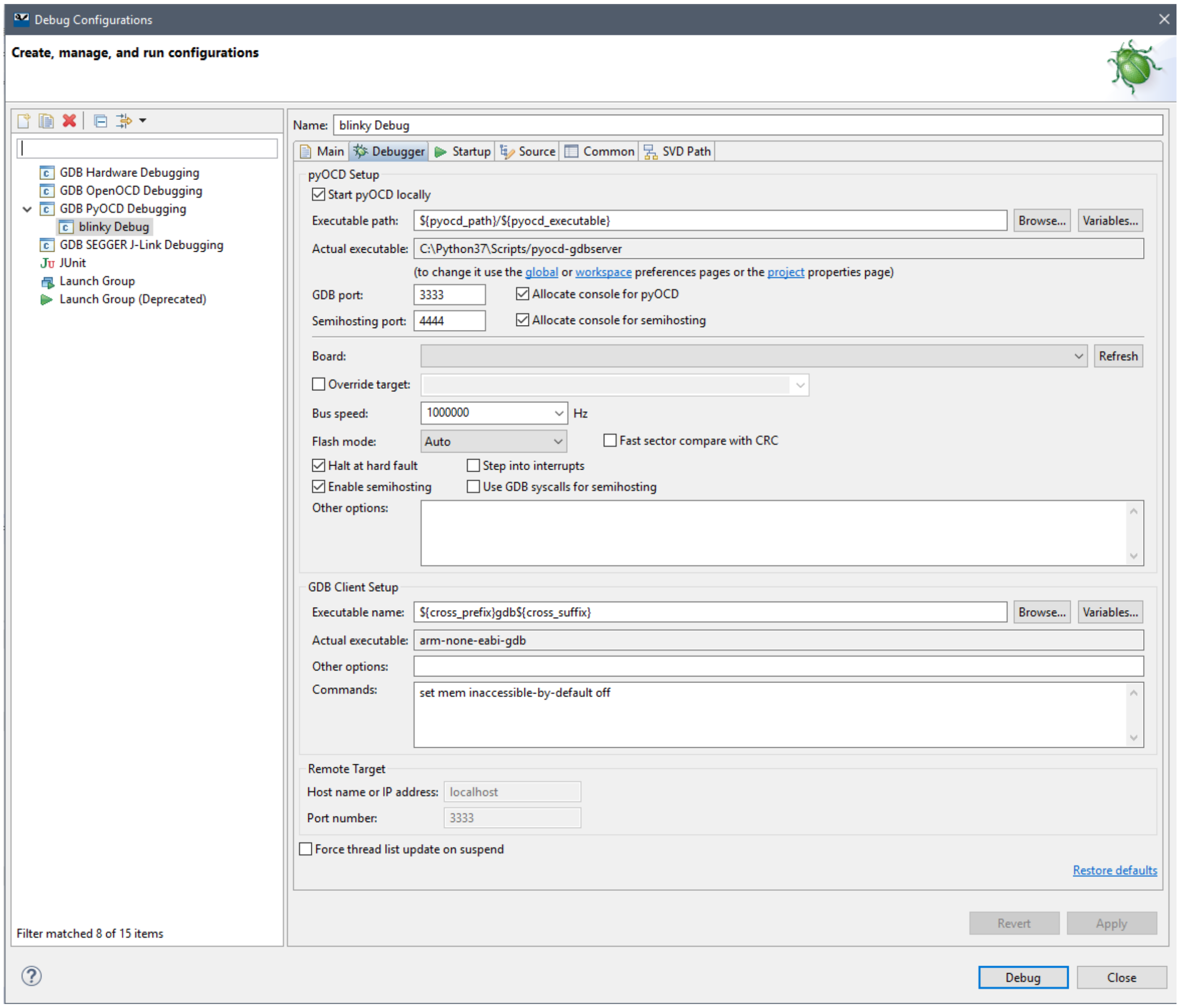
Build (if required) before launching

Build Configuration: Select Automatically

○ Enable auto build ● Disable auto build

○ Use workspace settings Configure Workspace Settings...

4. Configure the **Debugger** tab – all settings are default:



Create, manage, and run configurations

Name: blinky Debug

pyOCD Setup

☑ Start pyOCD locally

Executable path: ${pyocd_path}/${pyocd_executable}

Actual executable: C:\Python37\Scripts/pyocd-gdbserver

(to change it use the global or workspace preferences pages or the project properties page)

GDB port: 3333 ☑ Allocate console for pyOCD

Semihosting port: 4444 ☑ Allocate console for semihosting

Board:

☐ Override target:

Bus speed: 1000000 Hz

Flash mode: Auto ☐ Fast sector compare with CRC

☑ Halt at hard fault ☐ Step into interrupts

☑ Enable semihosting ☐ Use GDB syscalls for semihosting

Other options:

GDB Client Setup

Executable name: ${cross_prefix}gdb${cross_suffix}

Actual executable: arm-none-eabi-gdb

Other options:

Commands: set mem inaccessible-by-default off

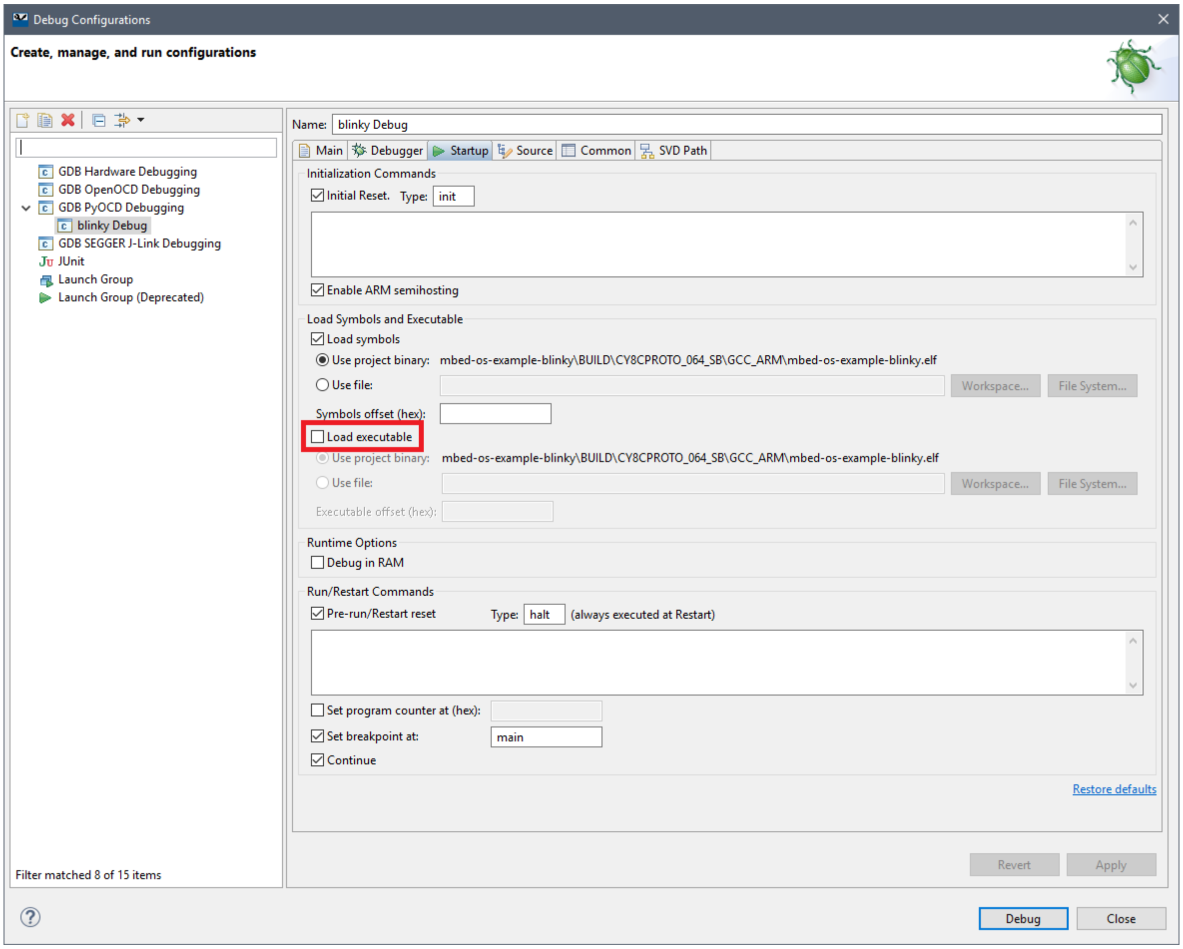
Remote Target

Host name or IP address: localhost

Port number: 3333

☐ Force thread list update on suspend

5. Configure the **Startup** tab: Unselect the **Load executable** check box.



Debug Configurations

Create, manage, and run configurations

- GDB Hardware Debugging
- GDB OpenOCD Debugging
- GDB PyOCD Debugging
  - blinky Debug
- GDB SEGGER J-Link Debugging
- JUnit
- Launch Group
- Launch Group (Deprecated)

Filter matched 8 of 15 items

Name: blinky Debug

Main | Debugger | Startup | Source | Common | SVD Path

Initialization Commands

☑ Initial Reset. Type: init

☑ Enable ARM semihosting

Load Symbols and Executable

☑ Load symbols

● Use project binary: mbed-os-example-blinky\BUILD\CY8CPROTO_064_SB\GCC_ARM\mbed-os-example-blinky.elf

○ Use file: Workspace... File System...

Symbols offset (hex):

□ Load executable

○ Use project binary: mbed-os-example-blinky\BUILD\CY8CPROTO_064_SB\GCC_ARM\mbed-os-example-blinky.elf

○ Use file: Workspace... File System...

Executable offset (hex):

Runtime Options

□ Debug in RAM

Run/Restart Commands

☑ Pre-run/Restart reset Type: halt (always executed at Restart)

□ Set program counter at (hex):

☑ Set breakpoint at: main

☑ Continue

6. Debug configuration is ready. Click **Apply**, then **Debug**.

Troubleshooting

- In case of messages like "unable to find device" execute "mbedls -m 1907:CY8CPROTO_064_SB", then check with "mbedls" if device is detected as CY8CPROTO_064_SB with code 1907.
- Keys, from ./keys folder is used for signing images by default, these keys should be used for provisioning.
- Consider using CyBootloader with “debug” mode in policy file. It produces logs, which are useful to understand whether CyBootloader works correctly.
- When running application with SMIF and _smif.json policy the field "smif_id" should be set to 1 for CY8CPROTO_064_SB.

4 ModusToolbox 2.0 – Provisioning Flow

This section shows how to provision the CY8CPROTO-064-SB kit in ModusToolbox 2.0 using CySecureTools.

Prerequisites

ModusToolbox 2.0 Installation

Install the ModusToolbox 2.0 software. Refer to the ModusToolbox Installation Guide.

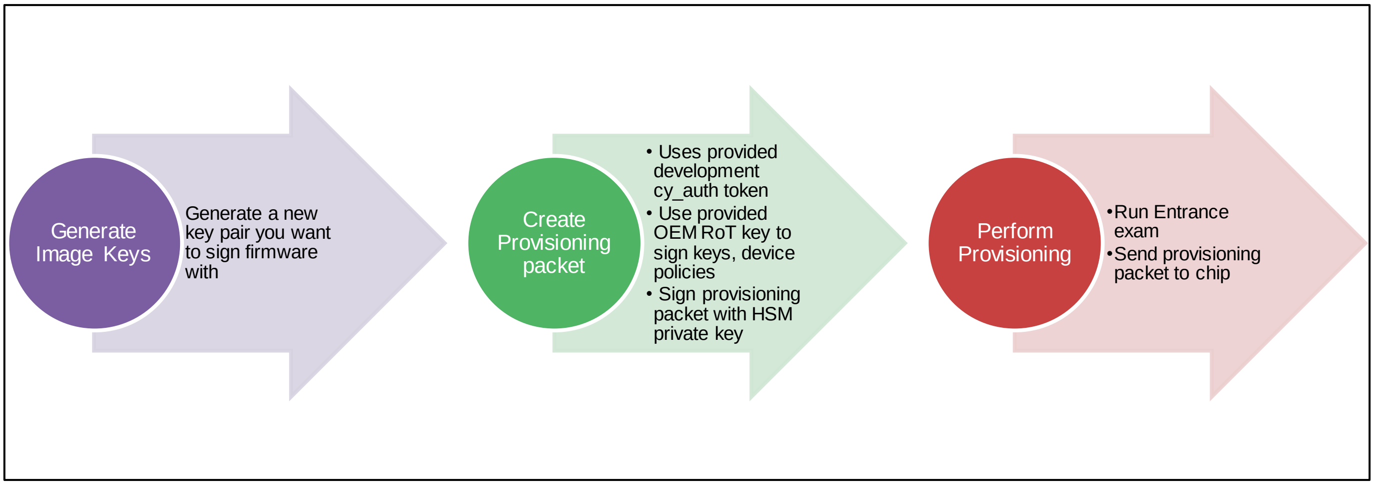
Note On Linux machines after installing ModusToolbox, run the *ModusToolbox/tools_2.0/modus-shell/postinstall* script.

CySecureTools Installation

Follow the instructions in the CySecureTools Installation and Documentation section.

Device Provisioning

For evaluation, the device provisioning flow can be done on your local development environment. For evaluation, a pre-signed development token is available in the SDK which authorizes a HSM key-pair provided in the SDK.


Generate Image Keys
Generate a new key pair you want to sign firmware with
Create Provisioning packet
• Uses provided development cy_auth token
• Use provided OEM RoT key to sign keys, device policies
• Sign provisioning packet with HSM private key
Perform Provisioning
•Run Entrance exam
•Send provisioning packet to chip

All the following steps should be executed from command line. Path to policy file (if uses custom policy) can be relative to current working directory or absolute. All paths to keys files inside policy file are absolute or relative to policy path.

For a detailed description of what the default policy file looks like, refer to the Understanding the Default policy section.

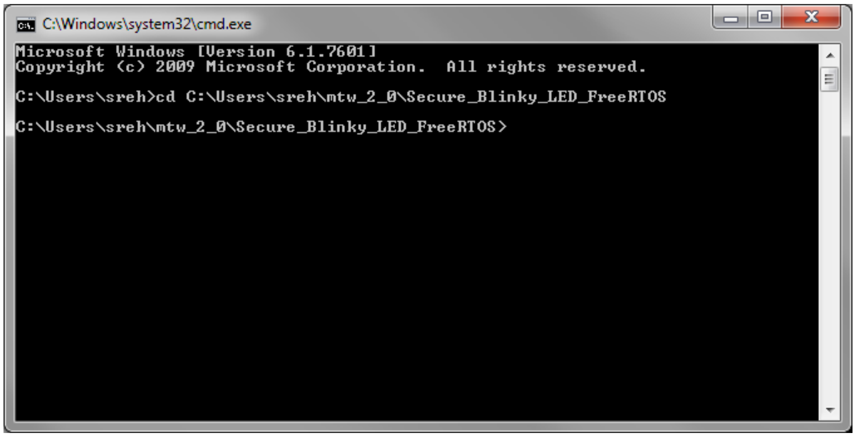
A. Create Blinkyled FreeRTOS Application Project

1. Launch ModusToolbox 2.0 IDE.
2. Open an existing workspace or create a new workspace
3. Accept the license agreement
4. Click on **File > New > ModusToolbox IDE Application**.
5. Select CY8CPROTO-064-SB kit and click on **Next >**.
6. Select "Blinky LED FreeRTOS" in "Starter Application" window and click on **Next >**.
7. Click on **Finish**. This may take a while as it pulls all required sources from respective repositories.

B. Setup policy file

Open your native command-line application and navigate to the *%WORKSPACE%/Secure_Blinky_LED_FreeRTOS\* directory;

For example in Windows7,



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\sreh>cd C:\Users\sreh\mtw_2_0\Secure_Blinky_LED_FreeRTOS

C:\Users\sreh\mtw_2_0\Secure_Blinky_LED_FreeRTOS>
```

Run the following command:

```
*Copy default policy from cysecuretools into ModusToolbox project*

*Example usage for the CY8CPROTO-064S1-SB kit*

python -c "import os; import shutil; import cysecuretools; os.makedirs('policy') if not
os.path.exists('policy') else None;
shutil.copy2(os.path.join(os.path.dirname(cysecuretools.__file__),
'targets/cy8cproto_064s1_sb/policy/policy_single_stage_CM4.json'),
'./policy/policy_single_stage_CM4.json')"
```

Note As an alternative, you can manually copy the policy file from your local python CySecureTools installation into the *%WORKSPACE%/Secure_Blinky_LED_FreeRTOS\policy* folder.

To use external memory (via SMIF) as staging (upgrade) area (slot_1) of the CM4 image, you should use *policy_single_stage_CM4_smif.JSON*.

C. Generate new keys

Note All key descriptions in the policy file should be done before calling key generation. The key generation function generates all keys from the policy file, including the encryption key.

Ensure you are in the “%WORKSPACE%/Secure_Blinky_LED_FreeRTOS/” directory and in command-line copy/paste the below command,

```
*Create keys, using specified policy:*

*Example usage for the CY8CPROTO-064S1-SB kit*

python -c "from cysecuretools import CySecureTools;tools = CySecureTools('CY8CPROTO-
064S1-SB', 'policy/policy_single_stage_CM4.json'); tools.create_keys();"

*Generic usage example*

python -c "from cysecuretools import CySecureTools;tools = CySecureTools('<TARGET>',
'policy/policy_single_stage_CM4.json'); tools.create_keys();"
```

Additional parameters for `create_keys` are `overwrite` and `out`. The parameter `overwrite` specifies behavior when `create_key` finds keys in the output path. The value `None` (is value by default) means that you will be asked about overwriting such keys or not. `True` means overwrite keys, `False` means keep existing. The parameter `out` specifies the output directory for keys and overwrites key paths from the policy file.

D. Create provisioning packets:

Ensure you are in the “%WORKSPACE%/Secure_Blinky_LED_FreeRTOS/” directory and in command-line copy/paste the below command,

```
*Create provisioning packets, using specified policy:*

*Example usage for the CY8CPROTO-064S1-SB kit*

python -c "from cysecuretools import CySecureTools; tools = CySecureTools('CY8CPROTO-
064S1-SB', 'policy/policy_single_stage_CM4.json'); tools.create_provisioning_packet();"

*Generic usage example*

python -c "from cysecuretools import CySecureTools; tools = CySecureTools('<TARGET>',
'policy/policy_single_stage_CM4.json'); tools.create_provisioning_packet();"
```

Note It is possible to choose between two variants of CyBootloader to be used during provisioning process – with or without diagnostic logs. It is recommended to use the default ‘debug’ mode, you can change this by editing the **cy_bootloader** section of the policy file:

```
CyBootloader without diagnostic logs
"cy_bootloader":
  {
    "mode": "release"
  }
CyBootloader with diagnostic logs
```

```
      "cy_bootloader":
        {
          "mode": "debug"
        }
```

E. Run entrance exam

Connect the kit to your PC.

ATTENTION The KitProg3 must be in CMSIS-DAP mode for provisioning. Press the ‘Mode’ button on the kit until the Status LED blinks slowly. For more details, refer to the KitProg3 User Guide.

In your command-line copy/paste the below command,

```
      *Run entrance exam *

      *Example usage for the CY8CPROTO-064S1-SB kit*

      python -c "from cysecuretools import CySecureTools; tools = CySecureTools('CY8CPROTO-
      064S1-SB', 'policy/policy_single_stage_CM4.json'); tools.entrance_exam();"

      *Generic usage example*

      python -c "from cysecuretools import CySecureTools; tools = CySecureTools('<TARGET>',
      'policy/policy_single_stage_CM4.json'); tools.entrance_exam();"
```

The Entrance exam is a test routine that does the following things:

- Verify if Device is in SECURE UNCLAIMED mode
- Verify if FlashBoot has not been modified/tampered
- Verify if User flash is empty and no code is running before any provisioning takes place

Failing the entrance exam returns an error in the command line. If there is any firmware running on the device, it can be erased using tools like the Cypress Programmer. The mbed-cli does not natively provide any commands to erase the chip. Note that the entrance exam is also automatically run before performing provisioning, so you can skip this step if needed.

E. Perform provisioning:

ATTENTION The PSoC 6 supply voltage of 2.5 V is required to perform provisioning. Refer to the relevant kit user guide to find out how to change the supply voltage of your kit.

Ensure you are in the *%WORKSPACE%/Secure_Blinky_LED_FreeRTOS/* directory. In your command-line copy/paste the below command for device provisioning:

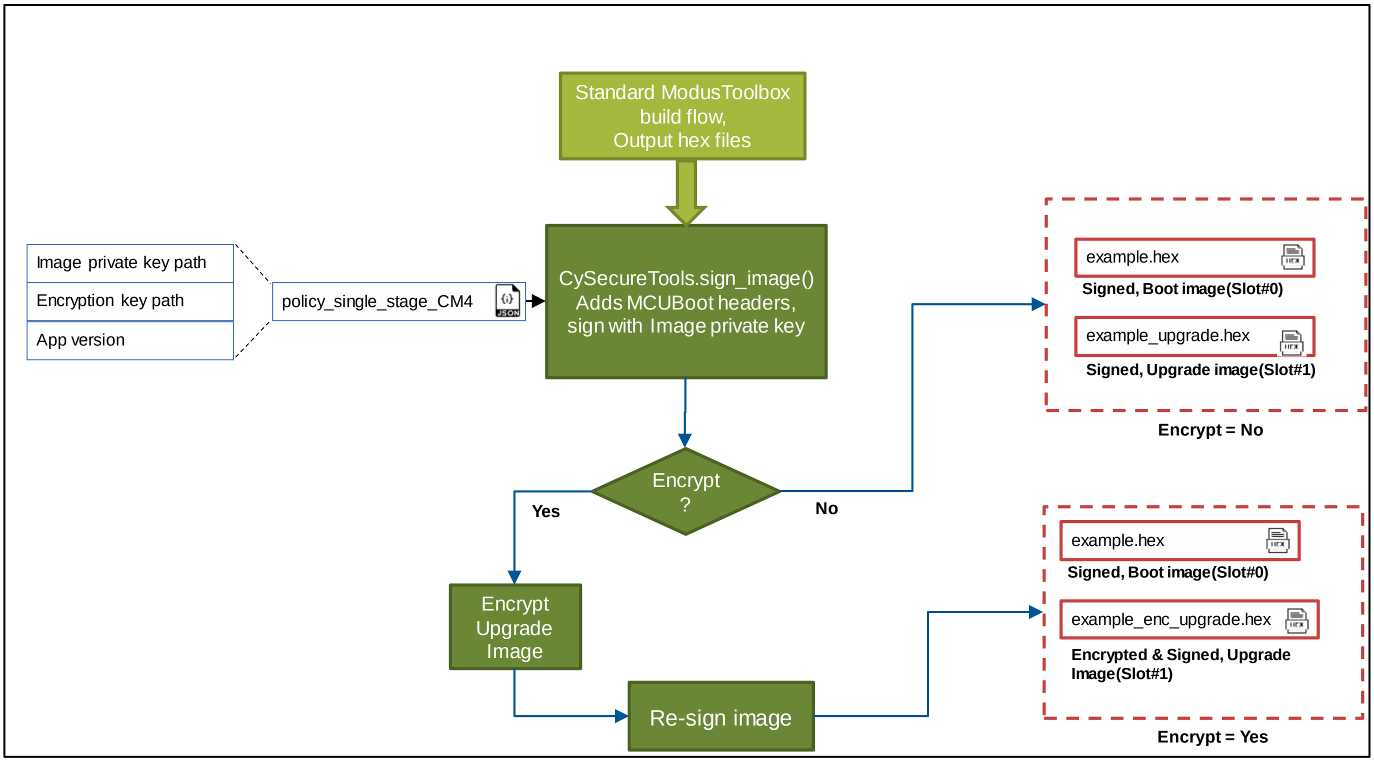
```
*Provision chip, using specified policy:*

*Example usage for the CY8CPROTO-064S1-SB kit*

python -c "from cysecuretools import CySecureTools;tools = CySecureTools('CY8CPROTO-
064S1-SB', 'policy/policy_single_stage_CM4.json'); tools.provision_device();"

*Generic usage example*

python -c "from cysecuretools import CySecureTools;tools = CySecureTools('<TARGET>',
'policy/policy_single_stage_CM4.json'); tools.provision_device();"
```

ModusToolbox Secure Image Generation

In ModusToolbox, PSoC 64 based kit targets have post-build signing scripts set up in the make file so the output binary is formatted and signed automatically according to the provisioned policy file; for example, *policy_single_stage_CM4.json*.

The post-build signing is part of the .mk file located in the target; for example, *\libs\TARGET_CY8CPROTO-064-SB\ CY8CPROTO-064-SB.mk*.

The following diagram shows the flow for signing and encryption using ModusToolbox:


Standard ModusToolbox build flow, Output hex files
Image private key path
Encryption key path
App version
policy_single_stage_CM4
CySecureTools.sign_image() Adds MCUBoot headers, sign with Image private key
Encrypt ?
Yes
No
Encrypt Upgrade Image
Re-sign image
example.hex
Signed, Boot image(Slot#0)
example_upgrade.hex
Signed, Upgrade image(Slot#1)
Encrypt = No
example.hex
Signed, Boot image(Slot#0)
example_enc_upgrade.hex
Encrypted & Signed, Upgrade Image(Slot#1)
Encrypt = Yes

The build process outputs two binaries:

- Signed boot image hex file. This is the exact binary that can be programmed to Slot#0 for the PSoC 64 to securely launch the application.

- Signed and encrypted(optional) update image hex file. This is the exact binary that can be programmed to Slot#1 for the PSoC 64 to perform a secure update and then launch the application.

Encrypting Generic Images

A generic HEX file (for example one that is produced by Mbed OS or any other build system) can also be converted into signed/encrypted image by executing the following command:

```
*Sign (and encrypt) image:*

python -c "from cysecuretools import CySecureTools;tools = CySecureTools('<TARGET>',
'policy/policy_single_stage_CM4.json'); tools.sign_image('path/to_hex_file.hex');"
```

Build and Run the Application

1. In the Project Explorer, right-click on the Blinky LED FreeRTOS project and select on **Build Project**.
2. Connect device to the computer over USB.
3. Right-click on Blinky LED FreeRTOS project and select **Run As > Run Configurations…**
4. On the dialog, select **GDB OpenOCD Debugging > Blinky LED FreeRTOS Program (KitProg3)** and click the **Run** button.

Debug the application

1. Right-click on Blinky LED FreeRTOS project and select **Run As > Debug Configurations…**
2. On the dialog, select **GDB OpenOCD Debugging > Blinky LED FreeRTOS Program (KitProg3)** and click the **Debug** button.

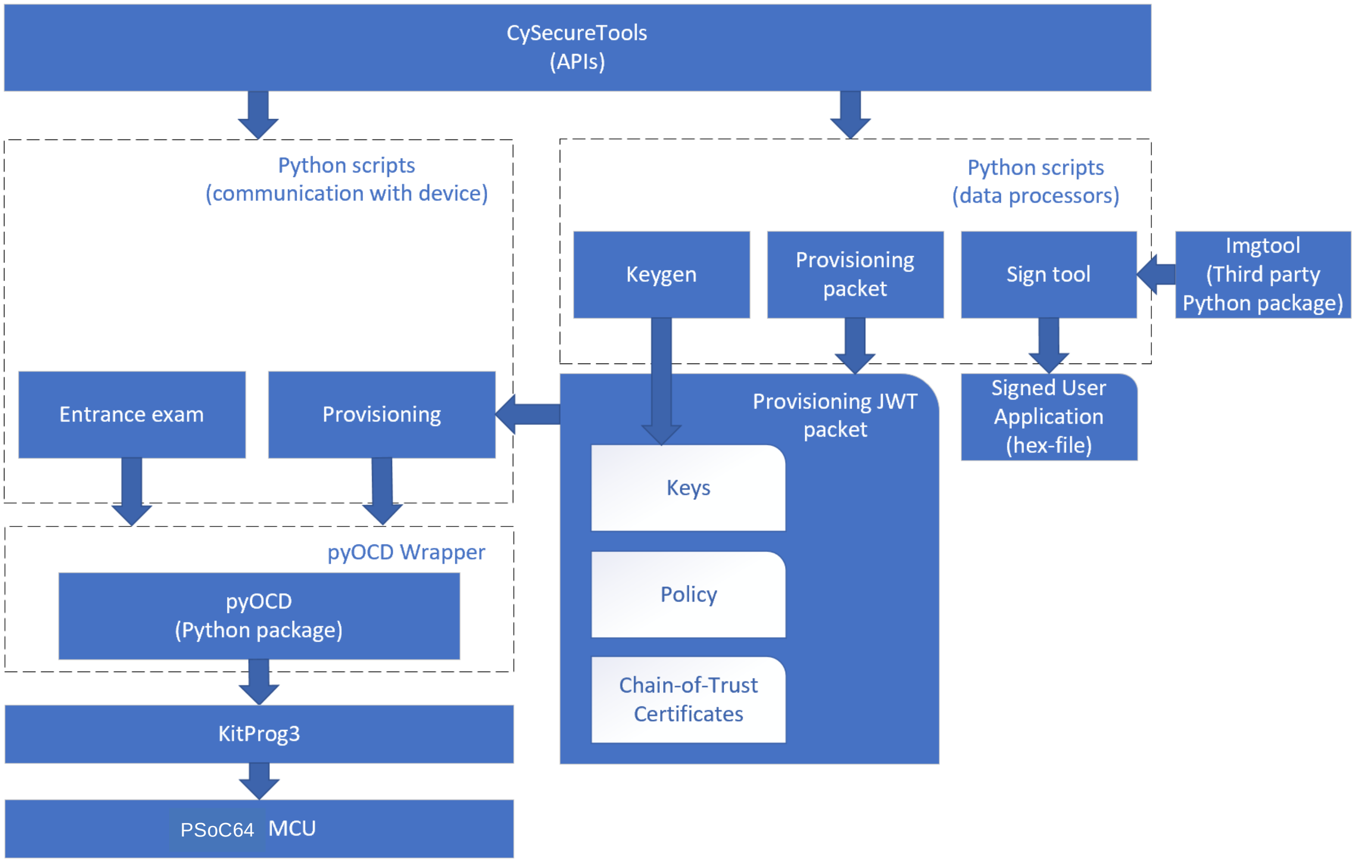
A Breakpoint is set at main function with default launch configurations. After the first step debugger breaks at main function.

5 CySecureTools Design

This section provides an overview of the CySecureTools python package design and details on the default policy

CySecureTools Component Diagram

The following diagram shows the high-level components of CySecureTools:

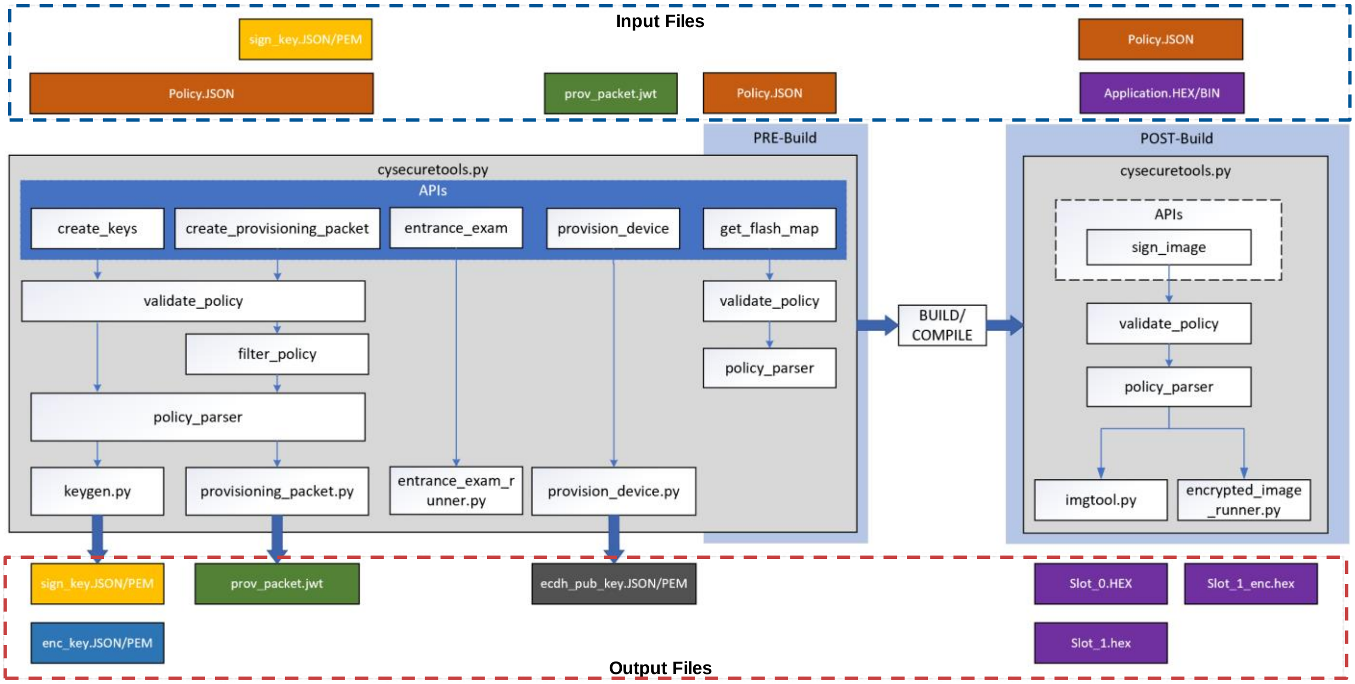


CySecureTools (APIs)
Python scripts (communication with device)
Entrance exam
Provisioning
pyOCD Wrapper
pyOCD (Python package)
KitProg3
PSoC64 MCU
Python scripts (data processors)
Keygen
Provisioning packet
Sign tool
Imgtool (Third party Python package)
Provisioning JWT packet
Keys
Policy
Chain-of-Trust Certificates
Signed User Application (hex-file)

CySecureTools Package Design

CySecureTools provides a wrapper over stand-alone scripts (on diagram, blocks with filenames *.py) that simplifies calling them with minimum number of arguments. However advanced users can use the scripts without wrapper and configure each argument as they need.

The various input, output files and python methods available are shown below,



A detailed view of the individual JWT structures is shown in the Provisioning JWT packet Reference section.

Understanding the Default policy

This section covers the details of the fields in the default policy provided in the CySecureTools. The contents can be classified as follows:

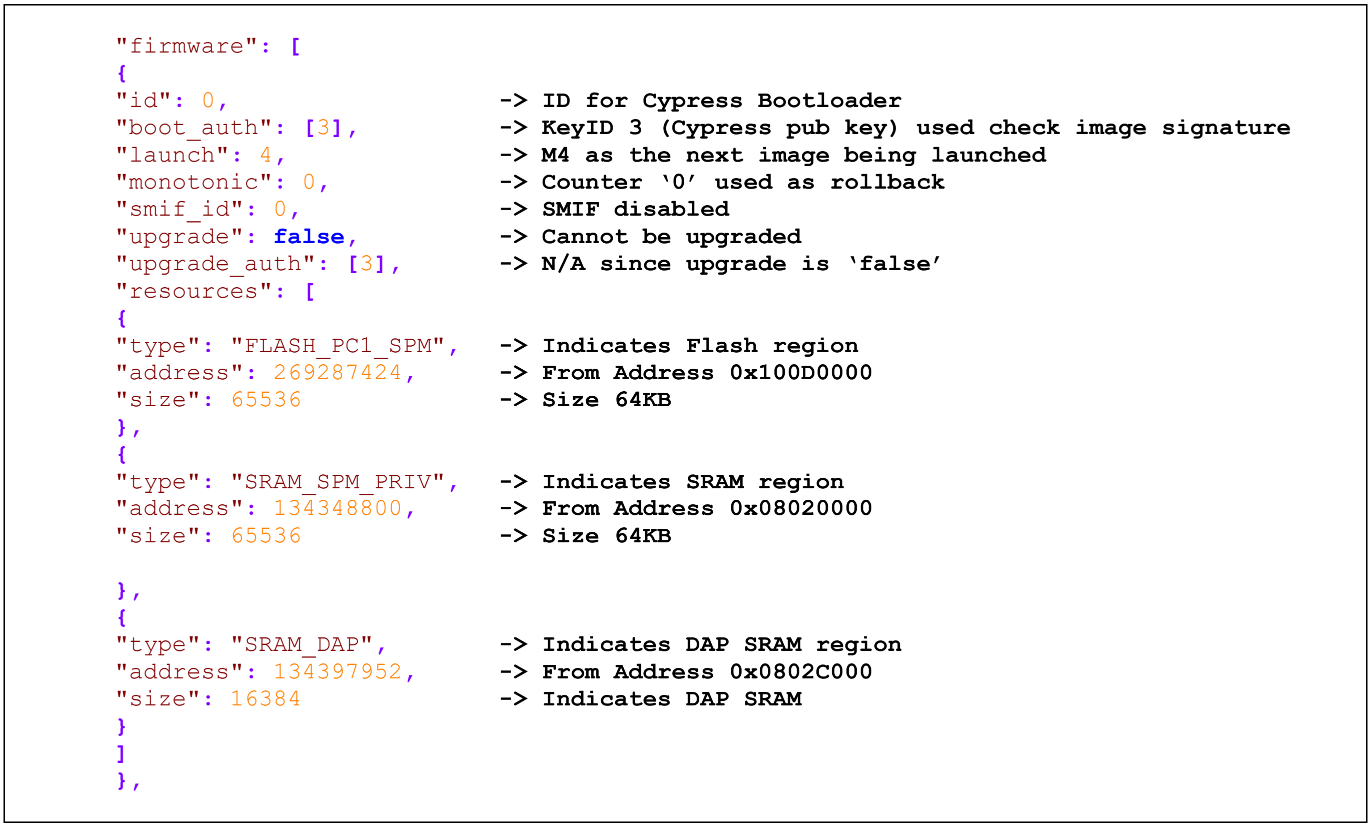
- Boot&Upgrade Policy
- Debug Policy
- Cypress Bootloader

Boot&Upgrade Policy

The Boot&Upgrade policy defines the memory regions and keys associated with images in the chip.

The following figures show the Cypress Bootloader settings in the default policy. These fields will never need to be modified, but they can be useful as a reference:

Figure 1: Cypress Secure Bootloader Boot&Upgrade policy



```
"firmware": [
{
"id": 0,                      -> ID for Cypress Bootloader
"boot_auth": [3],             -> KeyID 3 (Cypress pub key) used check image signature
"launch": 4,                  -> M4 as the next image being launched
"monotonic": 0,               -> Counter ‘0’ used as rollback
"smif_id": 0,                 -> SMIF disabled
"upgrade": false,             -> Cannot be upgraded
"upgrade_auth": [3],          -> N/A since upgrade is ‘false’
"resources": [
{
"type": "FLASH_PC1_SPM",      -> Indicates Flash region
"address": 269287424,         -> From Address 0x100D0000
"size": 65536                 -> Size 64KB
},
{
"type": "SRAM_SPM_PRIV",      -> Indicates SRAM region
"address": 134348800,         -> From Address 0x08020000
"size": 65536                 -> Size 64KB

},
{
"type": "SRAM_DAP",           -> Indicates DAP SRAM region
"address": 134397952,         -> From Address 0x0802C000
"size": 16384                 -> Indicates DAP SRAM
}
]
},
```

Figure 2: M4 Application image Boot&Upgrade policy

```
"firmware": [
{
"id": 4,                     -> ID for CM4 application

"boot_auth": [8],            -> KeyID 8 pubkey used to authenticate image

"monotonic": 0,              -> Counter ‘0’ used as rollback

"smif_id": 0,                -> External memory support disabled

"encrypt": false,            -> Don’t Encrypt image

"encrypt_key": "../keys/aes128.key",     -> AES key to encrypt image with

"encrypt_key_id": 1,                     -> Kid pubkey used for ECDSA extraction
    of image key

"upgrade": true,             -> Can be upgraded

"upgrade_auth": [8],         -> If upgradeable, use Kid 8 to check signature

"resources": [
{
"type": "BOOT",              -> Indicates Slot#0 for MCUBoot
"address": 268435456,        -> From Address 0x10000000
"size": 327680               -> Size 320KB
},
{
"type": "UPGRADE",           -> Indicates SRAM region
"address": 268763136,        -> From Address 0x10050000
"size": 327680               -> Size 320KB

}
]

"boot_keys": [               -> Specify public keys to inject

{
  "key": "../keys/USERAPP_CM4_KEY.json"        -> Specify path of public key

  "kid": 8,                  -> Bind KeyID 8 to USERAPP_CM4_KEY.json
}

"version": "0.1",            -> Image version

"encrypt_peer": "../keys/dev_pub_key.pem",     -> Public key of device
```

Debug Policy

The Debug policy specifies how various access ports are configured for the part.

Figure 3: Debug policy

```
" debug":
{
        "m0p" : {
      "permission" : "disabled",        -> CM0+ DAP port is disabled
"control" : "firmware",                 -> User Firmware can open CM0+ DAP port
"key" : 5                               -> N/A valid only if control is “certificate”
        },
        "m4" : {
            "permission" : "allowed",   -> CM4 DAP port allowed to be open
            "control" : "firmware",     -> User Firmware can open/close CM4 DAP port
            "key" : 5                   -> N/A valid only if control is “certificate”
        },
        "system" : {
            "permission" : "enabled", -> SysAP DAP port open after bootup
            "control" : "firmware",   -> User Firmware can open/close SysAP port
            "key" : 5,                -> N/A valid only if control is “certificate”
            "syscall": true,          -> SysAP allowed to do system calls
            "mmio": true,             -> SysAP allowed to do MMIO accesses
            "flash": true,            -> SysAP allowed to access User Flash
            "workflash": true,        -> SysAP allowed to access Work Flash
            "sflash": true,           -> SysAP allowed to access Supervisory Flash
            "sram": true              -> SysAP allowed to access SRAM
        },
        "rma" : {
            "permission" : "allowed",-> RMA mode allowed, requires signed token
            "destroy_fuses" : [      -> If transitioning to RMA, destroy efuse data
                {
                    "start" : 888,    -> Start of eFuse Bit(bit 888)
                    "size" : 136      -> Number of eFuse Bits to erase
                },
                {
                    "start" : 648,    -> Start of eFuse Bit(bit 648)
                    "size" : 104      -> Number of eFuse Bits to erase
                }
            ],
            "destroy_flash" : [      -> If transitioning to RMA, destroy flash data
                {
                    "start" : 268435456, -> Start of flash address(0x10000000)
                    "size" : 851968      -> Size(832KB)
                },
                {
                    "start" : 269483520, -> Start of flash address(0x100FFE00)
                    "size" : 16          -> Size(16 bytes)
                }
            ],
            "key" : 5           -> Public keyID used to validate RMA request token
        }
```

Cypress Bootloader

```
        " cy_bootloader":
        {
              "mode" : "debug", -> CySecureBootloader will emit debug logs over UART
        }
```

Provisioning JWT packet Reference

1. prov_cmd.jwt

The provisioning packet sent to the PSoC 64 Secure MCU is a single prov_cmd.jwt. The following shows this JWT structure:

Structure:

```
       {
         {
         "cy_auth": "………",
         "rot_auth": "………",
         "image_cert": "………",
         "prov_req": "………",
         "chain_of_trust": [],
         "type": "HSM_PROV_CMD"
       } sig: HSM_PRIV_KEY
```

Object	Description
cy_auth	Cypress Authorization JWT, authorizes the HSM public key.
rot_auth	OEM/User authorization JWT, authorizes the HSM public key.
image_cert	Cypress Secure Bootloader image JWT, used for sending a Cypress Secure Bootloader signature.
chain_of_trust	Holds an array of X.509 certificates.
type	Specifies the JWT type as a string.

2. cy_auth.jwt

Structure:

```
       {
         "auth": {},
         "cy_pub_key": {Cypress root pub key},
         "hsm_pub_key": {HSM pub key},
         "exp": {Expiry time},
         "iat": {Issue time},
         "type": "CY_AUTH_HSM"
       } sig: CYPRESS_ROOT_PRIV_KEY
```

Object	Description
auth	Can specify authorization limits, currently unsupported.
cy_pub_key	Cypress Root Public key in the JWK format.
hsm_pub_key	HSM Root Public key in the JWK format.
exp	Specifies when the token expires in UNIX time.
iat	Specifies when the token was issued.
type	Specifies the JWT type as a string.

3. rot_auth.jwt

Structure:

```
{
  "hsm_pub_key": {HSM pub key},
  "oem_pub_key": {OEM RoT pub key},
  "iat": {Issue time},
  "prod_id": {Product Name},
  "type": "OEM_ROT_AUTH"
} sig: OEM_RoT_PRIV_KEY
```

Object	Description
hsm_pub_key	HSM Root Public key in the JWK format.
oem_pub_key	OEM RoT Public key in the JWK format.
iat	Specifies when the token was issued
prod_id	The product string, specified by the user. Note that this MUST match prod_id in the prov_req.JWT.
type	Specifies the JWT type as a string.

4. prov_req.jwt

Structure:

```
{
  "custom_pub_key": [{Key1}, …],
  "boot_upgrade": {…},
  "debug": {…}
  "prod_id": "my_thing",
  "wounding": {}
} sig: OEM_RoT_PRIV_KEY
```

Object	Description
custom_pub_key	The array of customer public keys to be injected in the JWK format.
boot_upgrade	Boot and Upgrade Policy JSON.
debug	Debug policy JSON.
prod_id	The product string, specified by the user. Note that this MUST match prod_id in the rot_auth.JWT.

Object	Description
wounding	Reserved.

5. boot_upgrade.JSON

Special Note The boot_upgrade.json and debug.json are located in the *policy_single_stage_CM4.json* file in the sb-tools\prepare folder. You can edit these freely and run the provisioning scripts to form new prov_cmd.jwt packets to provision chips.

Structure:

```
      {
          "firmware": [
            {
              "id": [Integer Value],
              "boot_auth": [Integer Value],
              "launch": Integer Value,
              "monotonic": [Integer Value],
              "resources": [
                {
                  "address": Integer Value,
                  "size": Integer Value,
                  "type": [STRING VALUE]
                },
              ],
              "smif_id": Integer Value,
              "upgrade": Boolean Value,
              "upgrade_auth": [Integer Value]
            }, …
          ],
          "title": "upgrade_policy"
        }
```

Object	Description	Range of valid values
id	Image id. (0-15: Cypress reserved, >15: customer specific)	A range of integers can be specified, however only 0 and 4 are supported for the correct provisioning of a device. "0" : The first firmware image started from RomBoot/FlashBoot (i.e. the boot loader). "4": The M4 Boot Image.
boot_auth	Specifies key index to use for validating the signature. These signatures are all verified during boot.	Can be any integer public key >3. For Cypress Secure Bootloader, the auth is “3”. For the M4 image, this can be any number depending on key_id specified in the JWK format in the custom_pub_key fields.
launch	Specifies next image ‘id’ being launched	“4” is the only valid value for Cypress Secure Bootloader and the Single image bootloader case.
monotonic	Indicates the monotonic counter number associated with this image. During secure boot this counter value is compared with the current_version code in the image being booted. During upgrade this counter is incremented to the value from the image header of the upgrade image.	0~15. Counters can be rolled up by the system firmware using SysCalls.

Object	Description	Range of valid values
resources: address	Specifies the start address of the image	The valid flash range address. Only decimal values are allowed, e.g.: 268435456 -> 0x10000000
resources: size	Specifies the size of the image	The valid flash range size in bytes. Only decimal values are allowed, e.g.: 327680-> 0x50000 -> 320KB
resources: type	Specifies type of image	Only “BOOT” and “UPGRADE” are user-modifiable fields for the M4 image. “BOOT” -> Slot#0 “UPGRADE” -> Slot#1
smif_id	Specifies if external memory is used for placing Slot#1 image	“0” – SMIF is disabled. “1” – If the CY8CPROTO_064_SB target is used.
upgrade	Specifies if updating is allowed for this image id	‘true’ -> Upgrades are allowed. ‘false’ -> Upgrade is not allowed.
upgrade_auth	Specifies key index to use for validating the signature of the upgrade. Allows upgrades to be checked by a different key if necessary.	Can be any integer public key >3. For Cypress Secure Bootloader, the auth is “3”. For the M4 image, this can be any number depending on key_id specified in the JWK format in the custom_pub_key fields.

6. debug.JSON

Special Note boot_upgrade.json and debug.json are located in the *policy_single_stage_CM4.json* file in the sb-tools\prepare folder. The user can edit these freely and run the provisioning scripts to form new prov_cmd.jwt packets to provision chips.

Structure:

```
{
    "m0p" : {
        "permission" : " STRING VALUE ",
        "control" : " STRING VALUE ",
        "key" : [Integer Value]
    },
    "m4" : {
        "permission" : " STRING VALUE ",
        "control" : " STRING VALUE ",
        "key" : [Integer Value]
    },
    "system" : {
        "permission" : " STRING VALUE ",,
        "control" : " STRING VALUE ",,
        "key" : [Integer Value],
        "syscall": Boolean Value,
        "mmio": Boolean Value,
        "flash": Boolean Value,
        "workflash": Boolean Value,
        "sflash": Boolean Value,
        "sram": Boolean Value
    },
    "rma" : {
        "permission" : "STRING VALUE ",
        "destroy_fuses" : [
            {
                "start" : Integer Value,
                "size" : Integer Value
            }
        ],
        "destroy_flash" : [
            {
                "start" : Integer Value,
                "size" : Integer Value
            },
        ],
        "key" : Integer Value
    }
}
```

Object	Description	Range of valid values
m0p/m4/system: permission	Specifies the permission level for the associated DAP port.	“Enabled” – The DAP port is open after bootup. “Allowed” – The DAP port can be opened after bootup, see the “control” field. “Disabled” – The DAP port is closed after bootup.

Object	Description	Range of valid values
m0p/m4/system: control	Specifies how the DAP port can be opened after bootup. The field is only valid if “permission” is “Allowed”.	“firmware” – The code the user can choose to open the DAP port depending on some custom code. “certificate” – A signed token must be presented using a SysCall to open the DAP port.
m0p/m4/system: key	Specifies which Key Id to use for certificate validation in “control” field	The key ID must be >3, point to the key provisioned in the custom_pub_key field.
system: syscall/mmio/flash/workflash/ sflash/sram	Specifies which regions the SysAP port is allowed to access	“true” -> Access to the region is allowed. “false” -> Access to the region is not allowed .
rma: permission	Specifies if RMA is allowed	“Disabled” – RMA is not allowed. “Allowed” – The RMA stage is available and can be entered by presenting a certificate using key> to a SysCall API. The system will destroy fuse and flash contents as specified in <destroy_fuses> and <destroy_flash> before transitioning to RMA stage.
rma: destroy_fuses: start	Starting fuse bit number for region	0~65536. Check the part datasheet for the eFuse allowed address.
rma: destroy_fuses: size	Number of fuse bits in region	0~65536. Check the part datasheet for the eFuse allowed size.
rma: destroy_flash: start	Starting byte address of region (will be rounded down to nearest program/erase boundary)"	0~0xFFFFFFFF. Check the part datasheet for the flash allowed address.
rma: destroy_flash: size	Size in bytes of region (will be rounded up so region is integral number of program/erase units)	0~0xFFFFFFFF. Check the part datasheet for the flash allowed size.
rma: key	The key slot number of the key used to validate authorization to enter RMA stage	The key ID must be >3, point to the key provisioned in the custom_pub_key field.