# Contents

## 1 Introduction
- Where to Get the Secure Boot SDK ................................................................. 4
- Using this Guide ................................................................................................. 4
- Definition of Terms .......................................................................................... 4
- Revision History ............................................................................................... 5

## 2 Overview ......................................................................................................... 6
- Secure Boot SDK Components .......................................................................... 6
- What is Provisioning? ....................................................................................... 6
- Cypress Secure Bootloader .............................................................................. 11
- Secure Boot SDK Organization ........................................................................ 12

## 3 Example Provisioning Using Mbed OS ......................................................... 13
- Prerequisites ..................................................................................................... 13
- Device Provisioning .......................................................................................... 13
- Upgrade Images ................................................................................................ 16
- Testing Application Code .................................................................................. 18
- Debugging Application Code ........................................................................... 18

## 4 Provisioning Script Flow Details ................................................................. 23
- Provisioning Flow ............................................................................................. 23
- Provisioning JWT packet Reference ................................................................. 25
1 Introduction

Cypress provides the Secure Boot SDK to simplify using the PSoC64 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 with a private Cypress branch of Mbed OS. The branch will be merged in pull requests 11018 and 11046. For early access to this private branch, contact your Cypress Representative.

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.

## Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Description of Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>7/19/19</td>
<td>New document.</td>
</tr>
</tbody>
</table>
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:

- TLS authentication
- 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 contains the following components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provisioning Scripts</td>
<td>Python scripts for provisioning the PSoC 64 Secure MCU. Scripts are based on Kitprog3 CMSIS-DAP and will require the CY8CPROTO-064S1-KIT to test.</td>
</tr>
<tr>
<td>Entrance Exam Scripts</td>
<td>Runs an entrance exam on the PSoC 64 Secure MCU to ensure no tampering has occurred.</td>
</tr>
<tr>
<td>Cypress Secure Bootloader Image</td>
<td>The first stage bootloader based on an open source MCUBoot [1] library.</td>
</tr>
<tr>
<td>Sample Provisioning Tokens</td>
<td>Examples to be used as templates for provisioning tokens.</td>
</tr>
</tbody>
</table>

1 [1]: https://mcuboot.com/

What is Provisioning?

Provisioning is a process where 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.
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.

The following series of 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:

<table>
<thead>
<tr>
<th>PSoC64 at manufacturing</th>
<th>Taking over Root-of-Trust</th>
<th>PSoC64 with OEM pub key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trims, FlashBoot, unique device keys, hashes etc.</td>
<td>Trims, FlashBoot, unique device keys etc.</td>
<td>Trims, FlashBoot, unique device keys etc.</td>
</tr>
<tr>
<td>Cy Root Public Key</td>
<td>Cy Root Public Key</td>
<td>OEM RoT Public Key</td>
</tr>
</tbody>
</table>

PSoC64 verifies Cy signature, accepts HSM key validate OEM Key as valid (cy_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 will be 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 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 Provisioning script flow details.
Cypress Secure Bootloader

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. 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. Also, Cypress Secure Bootloader 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:

Cypress Secure Bootloader performs the following series of 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, then upgrades it
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.

**Note** The pre-built Cypress Secure Bootloader launches the CM4 core with PC=6 protection context.

**Special Note** 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 and blown in eFUSE. The boot code will check for a HASH match after that point and does not check for the signature again.

## Secure Boot SDK Organization

The following shows the default path of the SDK in Mbed OS:

```
<mbed>\targets\TARGET_Cypress\TARGET_PSOC6\sb-tools
```

The sb-tools folder contains the following sub-folders:

<table>
<thead>
<tr>
<th>Secure Boot SDK folder</th>
<th>Folder Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>sb-tools</td>
<td>Folder Descriptions</td>
</tr>
<tr>
<td>execute</td>
<td>Provisioning scripts Contains python scripts for provisioning.</td>
</tr>
<tr>
<td>imgtool</td>
<td>MCUBoot imagetool Contains a python script used by MCUBoot for formatting a built binary image into the MCUBoot format.</td>
</tr>
<tr>
<td>keys</td>
<td>Image key tools Contains tools for generating a new image key, as well as the image key used to sign applications.</td>
</tr>
<tr>
<td>packet</td>
<td>Output folder for tokens Contains JWTs generated by the provisioning scripts. Note that this is a generated folder that does not exist if you don't run any scripts.</td>
</tr>
<tr>
<td>prebuild</td>
<td>Required pre-requisite tokens and files Contains the Cypress Secure Bootloader files, a development authorization token the associated OEM RoT and HSM key pair. Editing these files will result in a failed provisioning process as the authorization flow breaks.</td>
</tr>
<tr>
<td>prepare</td>
<td>Input folder for policies Contains user-editable JSON for changing device security and boot&amp;upgrade policies</td>
</tr>
<tr>
<td>test</td>
<td>Test tools Contains scripts for testing the Secure Boot tools.</td>
</tr>
</tbody>
</table>
3 Example Provisioning Using Mbed OS

This section shows how to provision the CY8PROTO-064S1-SB kit. Note that these steps are also present in the following file:

<mbed>\targets\TARGET_Cypress\TARGET_PSOC6\sb-tools\README.md

Prerequisites

Python Installation

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. Install additional packages used for the Secure Boot SDK.

   Run the following command in <mbed>\targets\TARGET_Cypress\TARGET_PSOC6\sb-tools\n
   python -m pip install -r requirements.txt

Other Tool Requirements

- Fw-loader 2.2.138
- KitProg3 (DapLink) 1.11.237
- pyocd-0.14.1.dev685

Refer to the release notes installation instructions for these tools.

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.
All the steps run from the below folder,

<mbed>\targets\TARGET_Cypress\TARGET_PSOC6\sb-tools\

**NOTE:** If you are planning to run the example Blinky project, please follow steps 1, 2 in Testing a Mbed Blinky Application follow the steps in the following directory:

<Example Project>\mbed-os\TARGET_Cypress\TARGET_PSOC6\sb-tools\keys

1. **Generate new keys by executing the following commands from ./keys:**

   *Create keys for image signing:*
   ```
   python keygen.py -k 8 --jwk USERAPP_CM4_KEY.json --pem-priv USERAPP_CM4_KEY_PRIV.pem
   ```

   *Create key for image encryption:*
   ```
   python keygen.py --aes aes.key
   ```

2. **Create provisioning packets:**

   Use provisioning_packet.py from ./prepare folder.

   **Options:**

   ```
   --oem <filename>  OEM key file.
   --hsm <filename>  HSM key file.
   --cyboot <filename>  Cypress Bootloader image certificate.
   --cyauth <filename>  Provisioning authorization certificate.
   --policy <filename>  Policy file.
   --out <directory_path>  Output directory.
   --ckey <filename>  Customer key that will be used for image signing. Use the option multiple times to specify multiple keys.
   --devcert <filename>  Chain of trust certificate. Use the option multiple times to specify multiple certificates.
   ```
Example Provisioning Using Mbed OS

To create packet for CY8CPROTO_064_SB target:

```
python provisioning_packet.py --policy policy_single_stage_CM4.json --out ../packet --cyboot ../prebuild/CyBootloader_Release/CypressBootloader_CM0p.jwt --ckey ../keys/USERAPP_CM4_KEY.json
```

To use external memory (via SMIF) as staging(upgrade) area (slot_1) of CM4 image use policy file with corresponding name:

```
python provisioning_packet.py --policy policy_single_stage_CM4_smif.json --out ../packet --cyboot ../prebuild/CyBootloader_Release/CypressBootloader_CM0p.jwt --ckey ../keys/USERAPP_CM4_KEY.json
```

The prebuilt folder contains CyBootloader_WithLogs and CyBootloader_Release with corresponding *.hex and *.jwt files.
- WithLogs prints execution results to terminal.
- Release does not print to terminal and boots up silently.

**Note:** CypressBootloader_CM0p.jwt and CypressBootloader_CM0p.hex must be used in pair from the same directory in provisioning packet generation (.packets/prov_cmd.jwt) and provisioning procedure itself.

3. Run entrance exam

**ATTENTION:** The Kitprog must be in CMSIS-DAP mode for provisioning. The Status LED will be blinking slowly in this mode.

```
python entrance_exam_runner.py
```

This step ensures that the chip has not been tampered with and there is no existing firmware on the chip.

4. Perform provisioning:

**ATTENTION:** Proceed to Upgrade Images section first if UPGRADE image is needed.

**ATTENTION:** PSoC 6 supply voltage of 2.5V is required to perform provisioning

Execute `provision_device_runner.py`. The script will run with the default arguments if no arguments specified.

Default arguments can be overridden with a custom:

```
--prov-jwt <filename> Path to provisioning JWT file (packet which contains all data necessary for provisioning, including policy, authorization packets and keys)
--hex <filename> Path to Cypress Bootloader HEX binary file
--pubkey-json <filename> File where to save public key in JSON format
--pubkey-pem <filename> File where to save public key in PEM format
```
Example:

```python
python provision_device_runner.py --prov-jwt packet/prov_cmd.jwt --hex prebuild/CyBootloader_Release/CypressBootloader_CM0p.hex --pubkey-json keys/dev_pub_key.json --pubkey-pem keys/dev_pub_key.pem
```

**Upgrade Images**

Secure Boot enabled targets support image upgrades, if specified by policy. There are two types of upgrade images supported:

- signed, non encrypted
- signed, encrypted

The upgrade images types are determined by the following policy setting (firmware sections):

- "smif_id": 0, - 0 - SMIF disabled, 1 – Setup and enable SMIF hardware for CY8CPROTO-064S1-SB kit
- "upgrade": true/false, - should be set to true if UPGRADE supported, false - if disabled
- "encrypt": true/false, - should be set to true if encrypted UPGRADE supported, false - if disabled
- "encrypt_key_id": 1, - should remain unchanged, means that Device Key will be used in ECDH/HKDF protocol

**Requirements:**

The policy with _smif.json from prepare/ folder should be used.

**For encrypted image:**

- aes.key generated, as described in [Device Provisioning](#).
- dev_pub_key.pem must be placed in keys/ folder (this key is generated in provisioning procedure)
- secure_image_parameters.json file in the target directory must contain valid keys’ paths

**For non-encrypted UPGRADE image**

*Example policy for CY8CPROTO_064_SB:*

```
"smif_id": 1,
"upgrade": true,
"encrypt": false,
"encrypt_key_id": 1,
```

**Encrypted UPGRADE image:**

*Example policy for CY8CPROTO_064_SB:*

```
"smif_id": 1,
"upgrade": true,
"encrypt": true,
"encrypt_key_id": 1,
```
The modified policy file should be used for provisioning the device, as described in 4. Perform provisioning.

Now the Mbed OS application or test can be built as described in the TEST section. Images for UPGRADE are generated at build time, according to policy.

- Non encrypted UPGRADE image file name ends with upgrade.hex.
- Encrypted UPGRADE image file name ends with enc_upgrade.hex.

The upgrade image can be programmed onto the target board using DapLink. The upgrade procedure is performed after first reset.

**Encrypt generic image:**

The generic HEX file (for example one that is produced by mbed-os build system) can be converted into encrypted image by using encrypted_image_runner.py script located in sb-tools.

**Usage example:**

```python
python encrypted_image_runner.py --sdk-path . --hex-file someApplication.hex --key-priv keys/MCUBOOT_CM0P_KEY_PRIV.pem --key-pub keys/dev_pub_key.pem --key-aes keys/aes.key --ver 0.1 --img-id 3 --rib-count 0 --slot-size 0x50000 --pad 1 --img-offset 402653184
```

**Options:**

- `--sdk-path` - Path to Secure Boot tools folder
- `--key-priv` - ECC Private key used for image signing and for generating shared secret as per ECDH/HKDF.
- `--key-pub` - ECC Public key used for image signing and for generating shared secret as per ECDH/HKDF. Only device Key can be used in current implementation. It is generated by provisioning procedure.
- `--key-aes` - AES128 key and IV file raw image will be encrypted with.
- `--img-id` - Image ID of encrypted image. Must match one mentioned in policy for UPGRADE image.
- `--slot-size` - Slot_1 (UPGRADE) size. Must match one mentioned in policy for UPGRADE image.
- `--ver` - Version of image. Make sure it matches one defined in secure_image_parameters.json for a given HEX.
- `--rib-count` - Rollback counter. Make sure it matches one defined in secure_image_parameters.json for a given HEX.
- `--img-offset` - Starting address offset for UPGRADE image - passed as integer, as represented in policy.
Testing Application Code

Running Mbed tests

Build and run Mbed tests for CY8CPROTO_064_SB target with command:

```
  mbed test --compile -m CY8CPROTO_064_SB -t GCC_ARM -n tests-mbed* -v
```

Testing a Mbed Blinky Application

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

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

2. Replace the contents of the "mbed-os" folder with the private Cypress branch, in which all the provisioning steps were done on.

   **Note:** If the branch is already available in the Mbed OS public repository, you need not copy over the contents.

3. Build the application using the following command:

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

4. 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
   ```

Debugging Application 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 1.1.0.234 is installed on your machine.
- Python 3.7 is installed.
- pyOCD is installed.
- libusb driver is installed.

  □ For Windows:
Create Empty C/C++ Application

1. Open the ModusToolbox IDE.
2. Go to File->New->Project.
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 Main tab:
   a. Set Project to blinky.
   b. C/C++ Application points to the .elf file of the application.
   c. Check Disable auto build to prevent app building in IDE.
Example Provisioning Using Mbed OS

PSoC 64 Secure MCU
4. Configure Debugger tab – all settings are default
5. Configure **Startup** tab: Uncheck Load executable.

![Debug Configuration](image)

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 from CyBootloader_WithLogs folder. 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.
The provisioning flow consists of two distinct steps: a transfer of RoT and injection of assets. Yet, both steps are done in a single provisioning command to the PSoC64 chip. This section outlines the provisioning flow, JWT packets, and descriptions of the fields in the user-editable section.

### Provisioning Flow

#### Generating provisioning packets

The following shows a high-level flow and packets used by the provisioning_packet.py:

![Provisioning Flow Diagram](provisioning_flow_diagram.png)

A detailed view of the individual JWT structures is shown in Provisioning JWT packet Reference.
Device provisioning procedure

The following flowchart shows a high-level provisioning flow of the script (after forming JWT packets):

```
provisioning_device_runner.py

pyOCD Wrapper

Entrance Exam

CypressBootloader.HEX

Program CypressBootloader.HEX

prov_cmd.jwt

SIGN(HSM key)

ProvisionKeyAndPolicies

PSoC6 Device

Read Device Public Key (jwk)

dev_pub_key.json  dev_pub_key.pem
```
Provisioning JTW 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
```

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

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
```

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>auth</td>
<td>Can specify authorization limits, currently unsupported.</td>
</tr>
<tr>
<td>cy_pub_key</td>
<td>Cypress Root Public key in the JWK format.</td>
</tr>
<tr>
<td>hsm_pub_key</td>
<td>HSM Root Public key in the JWK format.</td>
</tr>
<tr>
<td>exp</td>
<td>Specifies when the token expires in UNIX time.</td>
</tr>
<tr>
<td>iat</td>
<td>Specifies when the token was issued.</td>
</tr>
<tr>
<td>type</td>
<td>Specifies the JWT type as a string.</td>
</tr>
</tbody>
</table>
3. rot_auth.jwt

Structure:

```json
{
    "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
```

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

4. prov_req.jwt

Structure:

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

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>custom_pub_key</td>
<td>The array of customer public keys to be injected in the JWK format.</td>
</tr>
<tr>
<td>boot_upgrade</td>
<td>Boot and Upgrade Policy JSON.</td>
</tr>
<tr>
<td>debug</td>
<td>Debug policy JSON.</td>
</tr>
<tr>
<td>prod_id</td>
<td>The product string, specified by the user. Note that this MUST match prod_id in the rot_auth.JWT.</td>
</tr>
<tr>
<td>wounding</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>
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:**

```json
{
   "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"
}
```

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
<th>Range of valid values</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Image id. (0-15: Cypress reserved, &gt;15: customer specific)</td>
<td>A range of integers can be specified, however only 0 and 4 are supported for the correct provisioning of a device. &quot;0&quot;: The first firmware image started from RomBoot/FlashBoot (i.e. the boot loader). &quot;4&quot;: The M4 Boot Image.</td>
</tr>
<tr>
<td>boot_auth</td>
<td>Specifies key index to use for validating the signature. These signatures are all verified during boot.</td>
<td>Can be any integer public key &gt;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.</td>
</tr>
<tr>
<td>launch</td>
<td>Specifies next image ‘id’ being launched</td>
<td>“4” is the only valid value for Cypress Secure Bootloader and the Single image bootloader case.</td>
</tr>
<tr>
<td>monotonic</td>
<td>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.</td>
<td>0~15. Counters can be rolled up by the system firmware using SysCalls.</td>
</tr>
<tr>
<td>resources:address</td>
<td>Specifies the start address of the image</td>
<td>The valid flash range address. Only decimal values are allowed, e.g.: 268435456 -&gt; 0x10000000</td>
</tr>
<tr>
<td>Object</td>
<td>Description</td>
<td>Range of valid values</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>resources: size</td>
<td>Specifies the size of the image</td>
<td>The valid flash range size in bytes. Only decimal values are allowed, e.g.: 327680- 0x50000 -&gt; 320KB</td>
</tr>
<tr>
<td>resources: type</td>
<td>Specifies type of image</td>
<td>Only &quot;BOOT&quot; and &quot;UPGRADE&quot; are user-modifiable fields for the M4 image. &quot;BOOT&quot; -&gt; Slot#0 &quot;UPGRADE&quot; -&gt; Slot#1</td>
</tr>
<tr>
<td>smif_id</td>
<td>Specifies if external memory is used for placing Slot#1 image</td>
<td>&quot;0&quot; – SMIF is disabled. &quot;1&quot; – If the CY8CPROTO_064_SB target is used.</td>
</tr>
<tr>
<td>upgrade</td>
<td>Specifies if updating is allowed for this image id</td>
<td>&quot;true&quot; -&gt; Upgrades are allowed. &quot;false&quot; -&gt; Upgrade is not allowed.</td>
</tr>
<tr>
<td>upgrade_auth</td>
<td>Specifies key index to use for validating the signature of the upgrade.</td>
<td>Can be any integer public key &gt;3. For Cypress Secure Bootloader, the auth is &quot;3&quot;. For the M4 image, this can be any number depending on key_id specified in the JWK format in the custom_pub_key fields.</td>
</tr>
</tbody>
</table>
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:

```json
{
    "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
    }
}
```

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