AIROC™ CYW20829 Bluetooth® LE: Designing a secured system

About this document

Scope and purpose

This document explains the security features of the Infineon's AIROC™ CYW20829 Bluetooth® LE and how to utilize these features to create a secured system for your application.

Intended audience

This document is intended for the users who want to learn about the security features of the AIROC™ CYW20829 Bluetooth® LE device.

Table of contents

Table of contents

1 Introduction

The Infineon AIROC™ CYW20829 MCU with the Bluetooth® LE connectivity is a Bluetooth® 5.4 core spec-compliant device for Internet of Things (IoT), smart home, and industrial applications. It establishes the right combination of low-power and performance for “always-on” applications. The AIROC™ CYW20829 Bluetooth® LE device is a programmable embedded system on chip.

For more information about the AIROC™ CYW20829 Bluetooth® architecture, refer to the device datasheet and related documentation.

This document discusses the following topics:

- Assessing the security needs
- Security features of the AIROC™ CYW20829 Bluetooth® LE MCU
- Understanding the AIROC™ CYW20829 Bluetooth® LE MCU bootup sequence
- The implementation of root of trust (RoT)
- Device lifecycle and transition.
- Signing, verifying, and encrypting your application.

The following are the key points of all topics that are discussed in this document:

- Provisioning decides whether CYW20829 will be a Secure silicon or Non-Secure by the Lifecycle user selects, this process is one-time and irreversible.
- You can configure different parameters in secure silicon such as policy and keys in one-time programmable memory (eFuse).
- Once the device is provisioned with set of keys, use the same key throughout the life of device.
- The CYW20829 silicon contains immutable BOOT_ROM code that manages initial security checks to boot L1 app (MCUBoot - Bootloader).
- BOOT_ROM uses signature verification and encryption (optional) of L1 app image as secure measure.
- L1 app (MCUBoot) can use different security parameters to verify and validate the L2 app (user app).
- You can generate keys required at various stages by Infineon-provided tool or any other tool.
- Boot flow => Reboot > BOOT_ROM validates/verifies L1 app > L1 app validates/verifies L2 app.

2 System security

To protect the hardware and the code (firmware) from being tampered, security need to be discussed at the beginning of any new project.

The products can be hacked via the following ways:

- **Direct access to the debug port:** With the use of common debug tools and dongles, you can reprogram the firmware or access the internal data.
- **Direct connection to a communication port:** Depending on the firmware, the direct connection to any of the communication ports (SPI, I2C, or a UART) may allow the firmware to be read or updated with a non-sanctioned software.
- **Network connections:** The firmware could be accessed with networks such as Bluetooth®, Wi-Fi, or Ethernet.
- **Third-party code:** Third-party code that is installed in the device after it has been shipped can also be used to access the firmware.

2.1 Basic definitions

Table 1 Definitions

Terminology	Description
Chain of trust (CoT)	Chain of trust is established by validating the blocks of software starting from the root of trust located in the ROM. The root of trust begins with the Infineon code residing in the ROM that cannot be altered.
Code signing	Process of calculating a hash of the code binary and encrypting the hash with a private key and appending this to the code binary.
Debug access port (DAP)	Interface between an external debugger/programmer and AIROC™ CYW20829 Bluetooth® for programming and debugging. This allows connection to one of the access ports (AP), CM33_AP.
Digital signature	Encrypting of the digest (hash of a data set). For example, the encrypted hash of the user application.
eFuse	One-time programmable (OTP) memory that by default is ‘0’ and can be changed only from ‘0’ to ‘1’. eFuse bits may be programmed individually and cannot be erased.
Flash (user)	Flash memory that is used to store your application code. It is non-volatile and can be reprogrammed.
Hash	A crypto algorithm that generates a repeatable but unique signature for a given block of data. This function is non-reversible.
IP	Intellectual property. This can be both code and data stored in a device.
Lifecycle stage (LCS)	The LCS is the security mode in which the device is operating. It has only four stages of interest: NORMAL, NORMAL_NO_SECURE, SECURE, and RMA.
Public key	When using asymmetrical cryptography such as RSA or ECC, a public key is used to validate firmware that was signed by the private key. It can be shared, but it should be authenticated or secured so it cannot be modified.
Private key	When using asymmetrical cryptography such as RSA or ECC, the private key is used to sign (encrypt the hash) of firmware after it is built but prior to being loaded into the device. It must be kept in a secure location, so it cannot be viewed or stolen.

System security

RMA	Return merchandise authorization
ROM	Read-only memory is non-volatile and is programmed as part of the fabrication process and cannot be reprogrammed.
BOOT_ROM	After a reset, the CPU starts executing code that has been programmed into ROM. This code cannot be altered and skipped from execution.
MCUBoot	MCUBoot is a secured bootloader for 32-bits microcontrollers.
Primary application (L1 app - Bootloader)	This may be the simple user application like “blinky” or can be a user bootloader like application which executes secondary application
Secondary application (L2 App)	The primary application verifies and validates the user application, and it can optionally encrypt the user application.
RSA-nnnn	An asymmetric encryption system that uses two keys. One key is private and should not be shared and the other is public and can be read without loss of security. The encryption/decryption is controlled by a key that is commonly 1024, 2048, or 4096 bits in length (RSA-1024, RSA-2048, or RSA-4096).
Serial memory interface (SMIF)	A SPI (serial peripheral interface) communication interface to serial memory devices, including NOR flash, SRAM, and non-volatile SRAM.
SHA-256	SHA-256 is a common cryptographic hash algorithm used to create a signature for a block of data or code. This hash algorithm produces a 256-bit unique signature of the data no matter the size of the data block.
AES	Advance encryption standard: it is used to encrypt a block or blocks of data with specified key
TRNG	True random number generator: unlike pseudo random number generator, output of TRNG is always random and cannot be hacked.
Table of contents 2	(TOC2) An area in external flash of the AIROC™ CYW20829 Bluetooth® that is used to store parameters and pointers to objects used for “Secure Boot”. Locations of one application pointer is stored here. The first pointer, Application1, must point to the first executable user code, which may be the bootloader or just the application.
JWT	JSON web token (JWT) is an open, industry standard (RFC 7519) method to securely represent claims between two parties.
OEM	Original equipment manufacturer: Although technically it means the Manufacturer, in this document it can be referred as the person or entity who has control over AIROC™ CYW20829 Bluetooth®
oem_pub_key_0, oem_pub_key_1	There are 2 keys with 16 bytes each which are stored in eFuse and are immutable (once written). Both key storages can be used to store OEM Public key hash. Referred as key ID ‘0’ and key ID ‘1’ further in document.
Sys-AP	System access port which enables you to debug and flash the memory.
CYW20829	Short representation of “AIROC™ CYW20829 Bluetooth® LE MCU”
Provisioning	Provisioning is an act of configuring the device with set of parameters which are essential in defining device’s behavior, such as SECURE/NON_SECURE, encrypted/non-encrypted and many more.

3 Security features

CYW20829 has several security features that are used to build a custom secured system. The combination of the following features can provide a secured system that meets the most security requirements.

- **eFuse block:** Important system values, security attributes such as configuration and keys are stored in the immutable memory block (one-time programmable).
- **Device lifecycle stage (LCS):** The LCS dictates how the device boots up and which security attributes to enable.
- **Root of trust:** This feature dictates that the validation of the user application code is linked all the way back to the device BOOT_ROM. A hash (SHA256) is calculated for the OEM public key and stored (first 128 bits) in eFuse. This hash is matched with the hash of an actual public key stored in an external flash. Any changes in the hash stored in eFuse or the OEM public key stored in the external flash will be detected and boot will fail.
- **"Secure Boot" sequence:** Depending on the LCS selected a secure or non-secure boot sequence can be initiated.
- **Debug Configuration:** Debug port can be configured to be enable/disable at the provisioning level and further a certificate-based authentication can be used to enhance security based on the lifecycle selected.
- **Secure firmware update:** Firmware can be updated over the air securely using an external tool like the MCUBoot (bootloader), which helps in verifying or validating the downloaded firmware.
- **Code signing:** Firmware is signed by public-private key pair and verified at BOOT_ROM stage.
- **Code encryption:** Firmware can be encrypted (optionally) and security is enhanced further.
- **Crypto:** Cryptographic hardware acceleration is used to save CPU time from complex crypto calculations

3.1 Keys (BOOT_ROM)

Only two keys (each 16-byte length) can be provisioned into eFuse data, *oem_pub_key_0* and *oem_pub_key_1*. The algorithm used for these keys is the RSA-2048 private-public key pair. CYW20829 provides an option to use two types of security level.

- Firmware Signature verification (RSA-2048).
- Encryption of firmware content (AES-128).

When using only the Signature verification process, *oem_pub_key_0* and *oem_pub_key_1* key storage space can be used to store the public key hash (*oem_pub_key_1* acts as a backup signing key hash can be used in case of reprovisioning). When image encryption is used, *oem_pub_key_1* storage space will be used to store the actual encryption key (128 bit) and *oem_pub_key_0* storage will be a hash of the public key(128 bit) used for signature verification.

3.1.1 Provisioning keys (OEM keys)

This private-public key pair (RSA-2048) can be generated by a provisioning tool (or by any other tool) and can be used for

- Signing the user application (private key).
- Validating signature of user application (public key).

This will be notified as the OEM private key and OEM public key further in the document. These keys are used only for provisioning (signing & verifying). It is generated in a private-public key pair in which the private key is

stored at a secure location decided by the OEM and the public key is stored on external flash and the hash of the public key is stored in eFuse.

3.1.2 Image encryption keys

This 128-bit AES key is generated by the provisioning tool (or by any other tool) and used ONLY for encrypting and decrypting the user image. For decrypting the user application by CYW20829, this key is stored in eFuse while provisioning and is immutable. This key can be used to encrypt/decrypt the primary application.

Store the key in a secure location as it is a key component to receive encrypted data (firmware/user app) over the SPI bus.

3.2 eFuse

eFuses are an integral part of security for the AIROC™ CYW20829 Bluetooth® devices. There are 1024 eFuse bits, which default to '0' and can only be programmed to a '1'. Once programmed, they cannot be erased back to '0' ever.

eFuse is programmed during provisioning of the device and the following parameters are configured:

- Debug policy
- Device policy (LCS)
- Hash of public key
- Encryption key for user application (optionally enabled)

Note: When programming the eFuse, the VDDIO0 pin must be connected to a 2.5 V supply.

3.3 Device lifecycle

The device lifecycle is a key aspect of the CYW20829's security. Lifecycle stages follow a strict irreversible progression dictated by programming the eFuses bits (changing a fuse's value from '0' to '1'). This system is used to protect the internal device data and code at the level required by the customer. These are the possible lifecycle of CYW20829 silicon. Lifecycle stages are governed by the LIFECYCLE_STAGE bits in eFuse and can only be advanced to the next lifecycle state as shown in the following figure. For example, once in the SECURE lifecycle state, the device can never return to the NORMAL state.

Security features

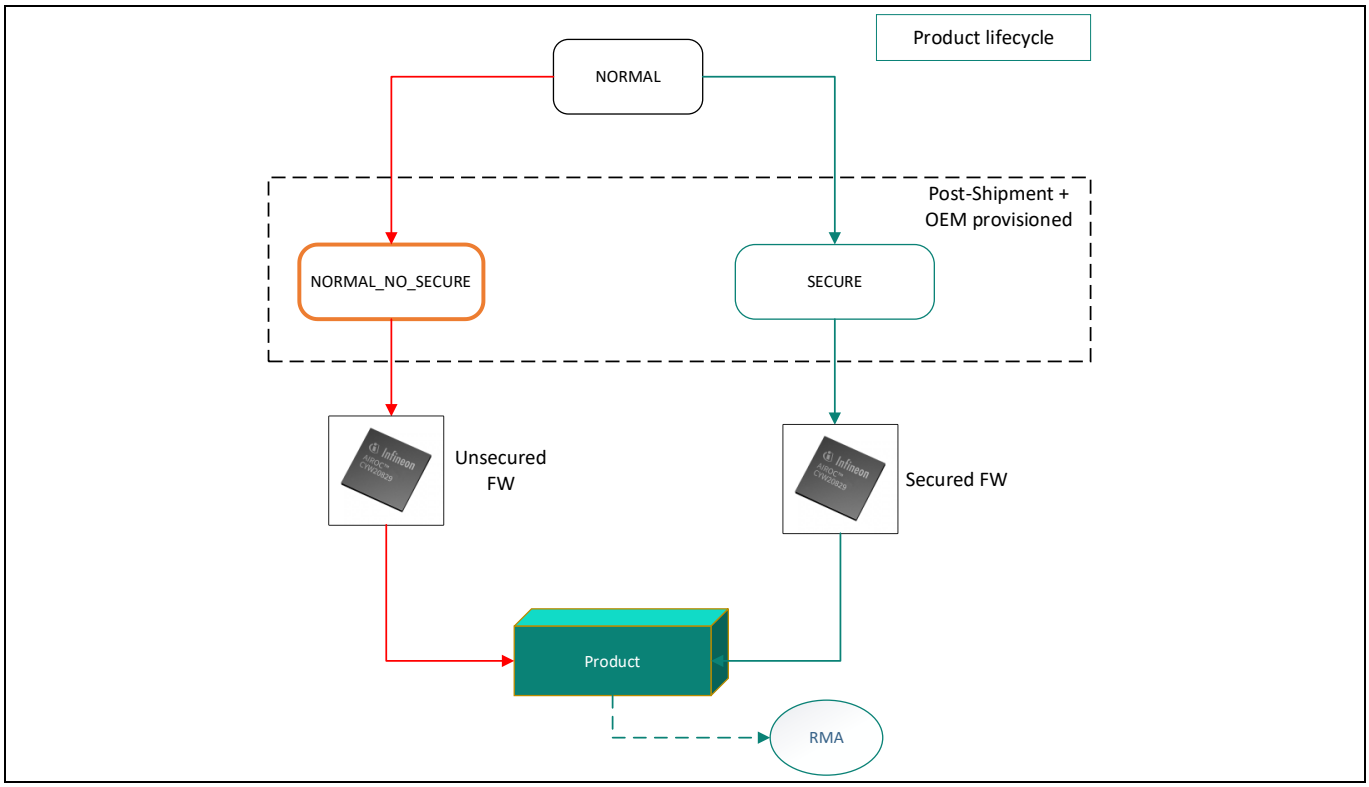

Figure 1 Device lifecycle

3.3.1 NORMAL

This is the initial lifecycle stage of the device. In this stage, the basic important parameters are written into the eFuse by Infineon. The device will be transitioned to the different lifecycle states as required.

3.3.2 NORMAL_NO_SECURE

This is the lifecycle stage in which the customers can use the silicon without the security features mentioned in this document. By default, you will have full debug access and may program certain areas in the eFuse using the Infineon tool. For more details, refer to the provisioning section.

3.3.3 SECURE

This is the lifecycle stage of a secured device. Before the transition to the SECURE lifecycle stage, the following tasks must be completed. Failure to do so could leave you with an inoperable device. More information on performing these steps is provided in the provisioning section of this document.

1. Make sure you have generated the private-public key pair that can be used by the tool provided by Infineon or generated by a third-party tool.
2. Use the Infineon tool to write the OEM public key hash into the eFuse.
3. Sign the firmware image (L1 App/Bootloader) using the OEM private key generated.
4. Transition to the SECURE lifecycle stage by using the proper policy mentioned in the Overview of implementation section.

In the SECURE lifecycle stage, the protection state is set to secure. A secured device will boot only when the authentication of its security parameters and application code succeeds.

Security features

After the MCU is in the SECURE lifecycle stage, it is irreversible. The debug ports may be disabled depending on your preferences, which means that there is no way to reprogram or erase the device with a hardware programmer/debugger. You should have a secondary application in case there are any upgrades required in which the primary application will remain stationary and will always be verified by the BOOT_ROM and the secondary application gets upgraded (for example, MCU_Boot (primary application) can be used to upgrade the user application/FW (secondary application)).

Note: *The code (user application) should be evaluated in the NORMAL_NO_SECURE lifecycle stage before the manufacturing devices moves to the SECURE lifecycle stage. This is to prevent a configuration error that could cause the part to be no longer accessible to the program, making it unusable.*

3.4 Root of trust

The BOOT_ROM implements a hardware root of trust (RoT) that provides an immutable trust anchor based on the authorization from Infineon or OEM. OEM authorization is required to provision the OEM keys and policies.

The key features of the hardware root of trust are:

- **"Secure Boot"**: At each stage of the boot process, 'BOOT_ROM' checks that the code is authorized to run before execution.
- **Lifecycle control**: RoT enforces debug/test access restrictions, upgrade restrictions, and lifecycle transition independent of any other trusted/untrusted firmware on the device. This enables Infineon to enforce the commercial agreements, and the OEM to enforce lifecycle-related security policies.
- **Secure debug**: RoT enforces debug control based on a debug policy that supports an authenticated debug certificate.
- **Secure provisioning**: RoT supports provisioning the device with an authorized set of keys, certificates, policies.

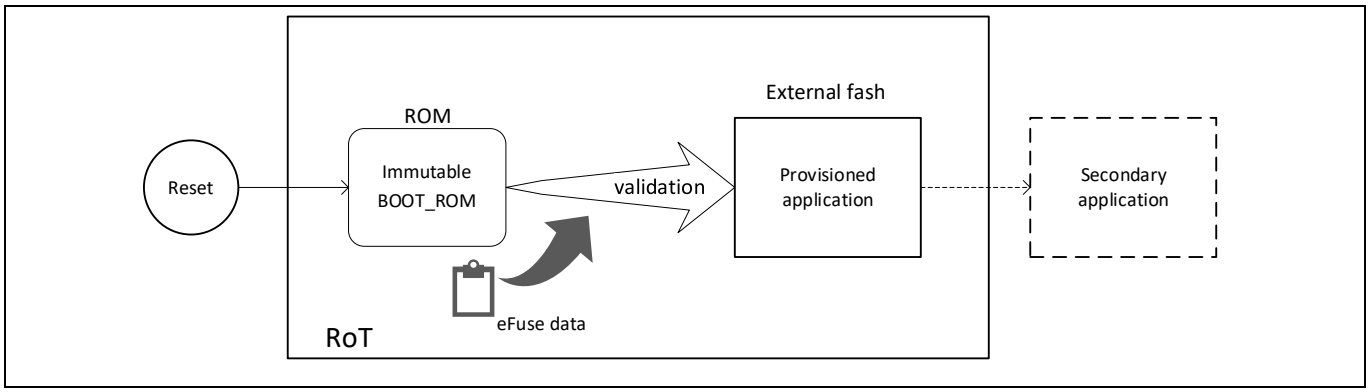

Figure 2 Root of trust

3.5 Debug

During all the device lifecycles, there are stages when tests and debug processes would be executed. During running the tests and debugging, it is important to reduce the possibility to leak important and sensitive information. Reducing can be done by analysis of possible paths (scenarios/instances) and covering the sensitive information using software/hardware protection.

Debug can be done on the following components.

- M33 CPU
- Sys-AP

You can debug only M33-related firmware (user app) by using the debug certificates. The generation of this certificate is explained in the provisioning document: “CySecureTools”. Debugging the other M33 core is restricted. If debug ports are permanently disabled while provisioning, you can no longer flash/debug the application henceforth. In the NORMAL_NO_SECURE LCS, the debug certificate is not required to be generated separately. However, in the SECURE LCS, you must generate a debug certificate (Mandatory) for the specific device for flashing/debugging.

3.6 Secure update/boot

“Secure Boot” is defined as the process of verifying and validating the integrity and authenticity of the existing and updateable firmware and software components as a prerequisite to their execution.

A “Secure Boot” process verifies the integrity and authenticity of executable code in a chain of trust starting from the BOOT_ROM”.

Software image integrity and authenticity validation are achieved by signing the image with asymmetric keys (RSA2048), and signature validation before execution. Stored computed cryptographic hash value of the image along with the OEM public key and signature validates the firmware (user app/image) on an external flash. The secure update process follows MCUBoot operation with image encryption optionally enabled. Check the MCUBoot documentation for more details.

In the AIROC™ CYW20829 Bluetooth® architecture, a “Secure Boot” chain of trust starts at the BOOT_ROM.

Security features

3.7 Crypto IP

AIROC™ CYW20829 Bluetooth® has an inbuilt hardware cryptographic accelerator, which reduces the load on the CM33 CPU for the crypto calculations. BOOT_ROM has access to this block and has drivers (library) to invoke commands related to it. SHA256, AES128, and TRNG blocks are present, which takes care of secure hash calculation and verification of the digital signature. Encryption/decryption: AES-128 hardware accelerator supports the following modes:

- Electronic code book (ECB)
- Cipher block chaining (CBC)
- Cipher feedback (CFB)
- Output feedback (OFB)
- Counter (CTR)

Hashing

- Secure Hash Algorithm (SHA-256) hardware accelerator.

Message authentication functions (MAC)

- Hashed Message Authentication Code (HMAC) acceleration using the SHA-256 hardware block.

AES-128

AES-128 component accelerates the block cipher functionality. This functionality supports forward encryption of a single 128-bit block with a 128-bit key.

SHA256

SHA-256 component to accelerate hash functionality. This component supports message schedule calculation for a 512-bit message chunk and processing of a 512-bit message chunk.

Vector unit

VU components to accelerate asymmetric key cryptography (for example, RSA and ECC). This component supports large integer multiplication, addition, and so on.

True random number generator (TRNG)

TRNG component based on a set of ring oscillators. The TRNG includes a hardware health monitor. It assures that the number generated is an actual random number and its repetition cannot be predicted/hacked.

For more details, see the AIROC™ Bluetooth® Low Energy 5.4 MCU datasheet.

External flash

AIROC™ CYW20829 Bluetooth® does not contain any on-chip flash memory, it requires external memory to run the user application. The SMIF (or QSPI) interface implements the SPI communication to external serial memory devices, including the NOR flash, SRAM, and non-volatile SRAM. The Infineon SMIF driver provides an easy-to-use interface to initialize and communicate with an external memory device through a single/dual/dual quad/quad SPI interface. Up to four memory devices can be connected using the slots.

Flash memory configurations are stored on the eFuse (while provisioning) to successfully run apps in external flash.

Security features

The table of content 2 (TOC2) resides in the external memory and stores the required data for the BOOT_ROM to launch the next application image and store the debug certificate. The TOC2 is placed at the fixed address 0x0000_0000 in the external memory so BOO_TROM knows where to get TOC2. The objects that are referenced using the TOC2 are verified by the BOOT_ROM before using them so there is no need to protect the TOC2 content by hash or something else. Also, the TOC2 referenced objects are not included in any hash calculations.

When the provisioned device is powered on, it reads the eFuse parameters, and depending on the policy it either boots into the no-secure or secure environment. The following flowchart explains the general flow.

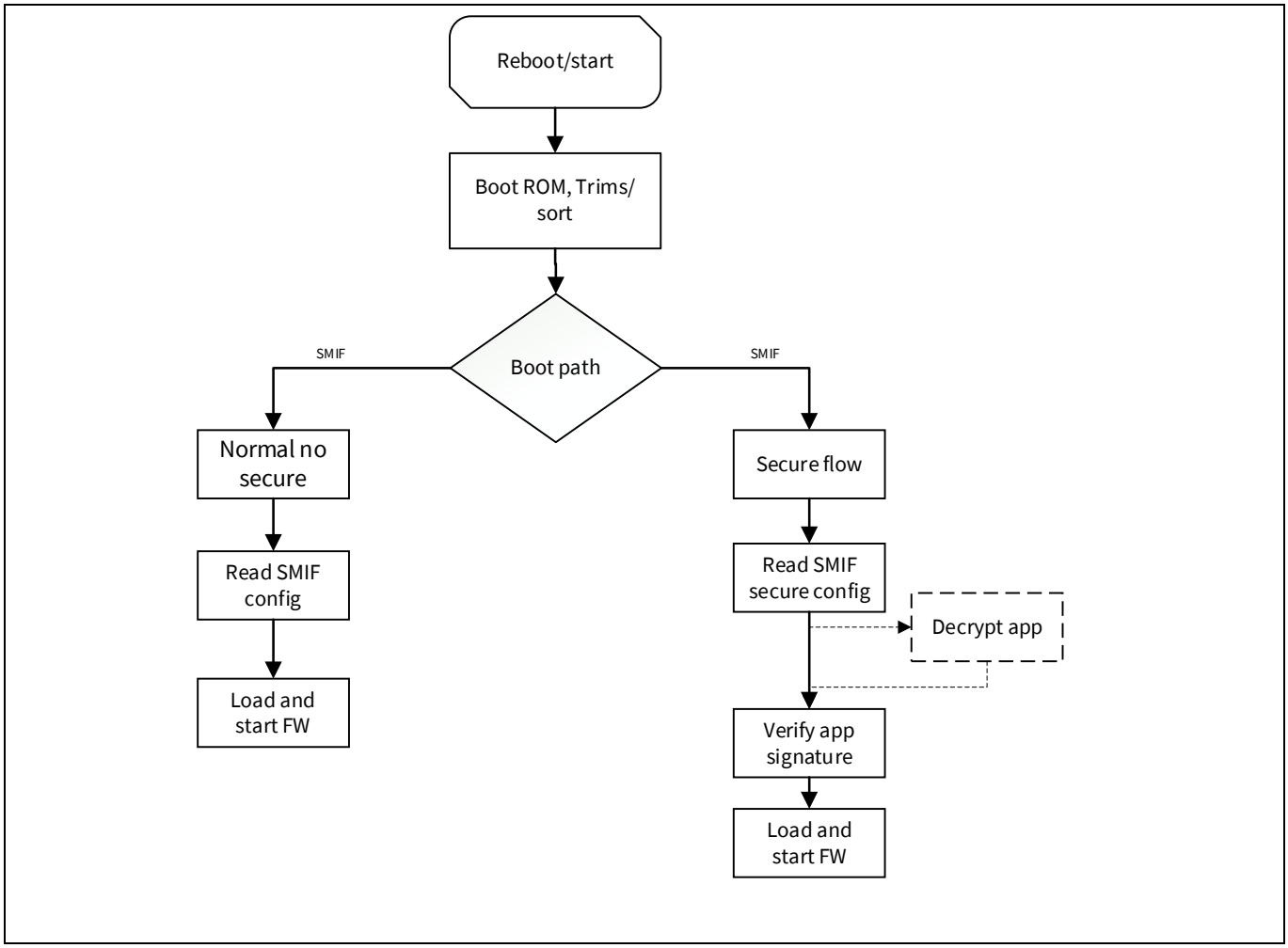

Figure 3 Boot flow

4 Overview of implementation

This section describes the various components of the 'Secure' system such as the tool required to execute secure methods on the application that is going to be implement on CYW20829 and the overall usage of the final L2 image to be flashed.

4.1 CySecure tool

"CySecureTools" is an Infineon provided, Python package that allows to perform policy editing, device provisioning, creating, and reading keys, signing user application, creating certificates and so on. Commands mentioned in this document are sample commands and the actual command format may change according to the user device configuration requirement.

Provisioning of the tool consists of:

- An example policy file in the JSON format.
- Generation of a signed final JWT or binary packet with policy and keys (to be provisioned in a device).
- Scripts to perform provisioning: these perform provisioning of several keys and policies, and create key files for the image signing tool.

The "CySecureTools" package contains the following policy types for the AIROC™ CYW20829 Bluetooth® target:

- **SECURE**: This is used to provision the device with the key(s) and convert it to the SECURE LCS. Filename: *policy_secure.json*.
- **NO_SECURE**: This is used to provision the device without key(s) and convert it to NORMAL_NO_SECURE LCS. Filename: *policy_no_secure.json*.
- **REPROVISIONING_SECURE**: This is used to reprovision the secured device (that was previously provisioned with a SECURE policy type). Filename: *policy_reprovisioning_secure.json*.
- **REPROVISIONING_NO_SECURE**: This is used to reprovision the non-secure device (that was previously provisioned with NO_SECURE policy type). Filename: *policy_reprovisioning_no_secure.json*.

When reprovisioning for NO_SECURE or SECURE LCS, only a limited number of parameters can be configured. Check the respective *policy.json* file and configure parameters accordingly. Only a device that is provisioned before can be reprovisioned.

4.2 Provisioning

Provisioning is the act of configuring a device with a Lifecycle that decides NORMAL_NO_SECURE or SECURE usage and can be configured with an authorized set of keys, certificates, credentials.

The provisioning happens using the provisioning application that is provided by Infineon. This application programs the OEM assets in the eFuse and advances the device LCS to production SECURE or NORMAL_NO_SECURE. The device is ready to launch the user application in these LCSs.

The provisioning applications require an SWD interface and require a physical connection.

Provisioning application is verified by the BOOT_ROM by validating its integrity and authenticity before executing it. The application format includes all the required metadata to verify. The provisioning flow is host driven; this flow is implemented in the "CySecureTools", which supports OEM provisioning.

Overview of implementation

Note: *Understand the procedures mentioned on the "CySecureTools" webpage before executing any command. Prior knowledge of ModusToolbox™ is required.*

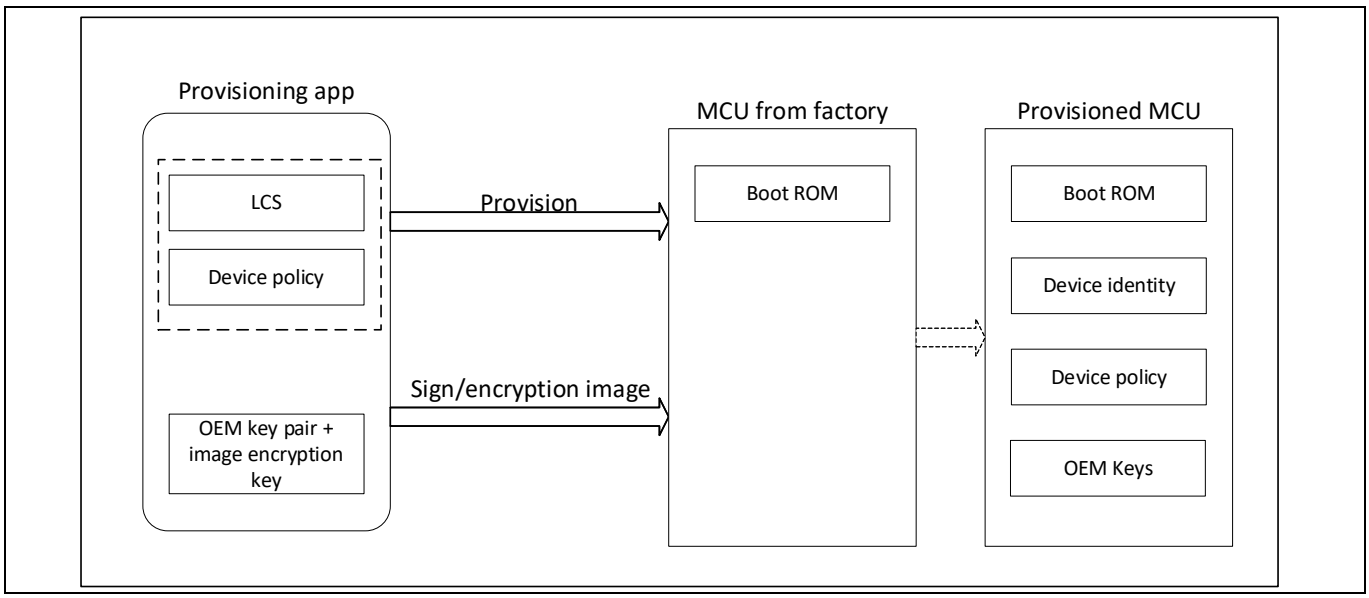

Figure 4 Provisioning flow

4.3 Image encryption

For the image encryption, "CySecureTool" uses the AES algorithm with a direct use of a 128-bit key, which is provisioned to the device. The same key must be used for the image encryption.

This key is provisioned into the device by storing it into eFuse. If you use image encryption, key ID 1 space is used to store this encryption key. The user application image may consist of multiple images (for example, MCUBoot application, L1 user application, and next user application).

Each image may have its own TOC and application descriptor data. However, during the encryption or decryption process, the boot ROM considers MCU_Boot as the primary image. First, TOC2, and descriptor are used to get the MCU_Boot image (primary application) start address. Contents of the MCU_Boot/primary application are encrypted.

It is the primary application's task to execute the next image. For more information, see the MCU_Boot CYW20829 guidelines.

Note: *The image encryption step is optional but encouraged to use as it enhances and completes the secure implementation.*

Overview of implementation

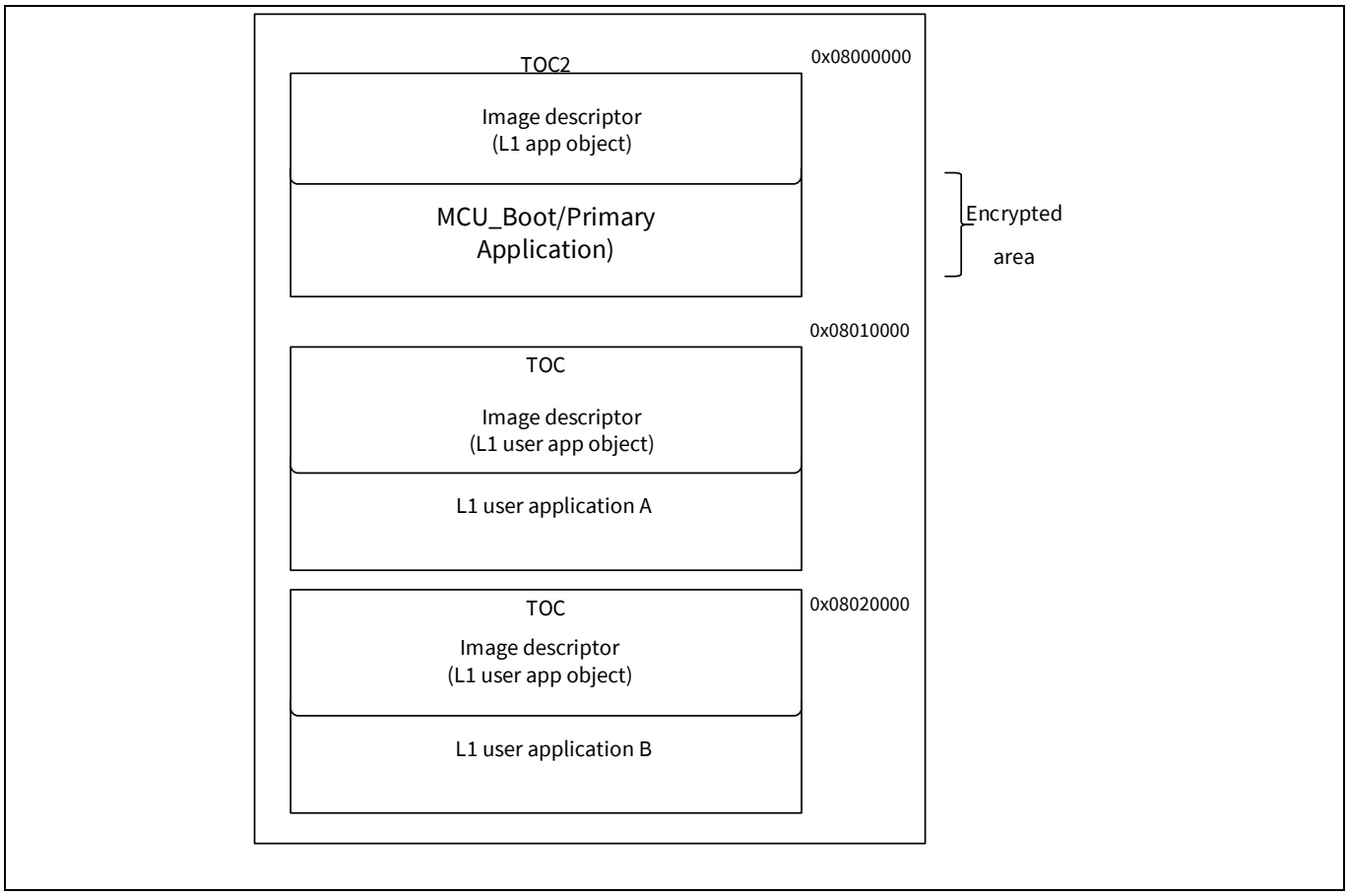

Figure 5 Image encryption

A 128-bit encryption key is provisioned as is. There is an option to have two OEM keys (*oem_pub_key_0* and *oem_pub_key_1*). However, if the encryption is used, only *oem_pub_key_0* can be used. The encryption key will be placed instead of the second OEM key hash.

4.4 Debug certificate

The debug certificate is used by BOOT_ROM to enable the CM33-AP and/or Sys-AP when it is temporarily disabled.

Note: The certificate cannot enable an access port that is permanently disabled by the access restrictions. Also, the debug certificate can be used to enable/disable invasive or non-invasive debug for CM33-AP.

The command creates a debug or RMA certificate binary based on the template provided in “CySecureTools”. The certificate must contain an OEM public key for further verification. If it is signed using the local private key, then the public key is extracted from the private key.

If the certificate is going to be signed using the hardware security module (HSM), create a non-signed certificate and specify the public key. While flashing “secure” CYW20829, a debug certificate is mandatory.

BOOT_ROM validates the debug certificate in the boot flow; upon authentication, it starts secure debugging.

4.5 Final image

After creating a signed image and optionally encrypting it, you need a complete image to be flashed into the external flash. Use the normal image created by the build process and use it along with the TOC2 + L1APP header as the final image. Figure 6 shows the flowchart that demonstrates the blocks involved in creating a final image.

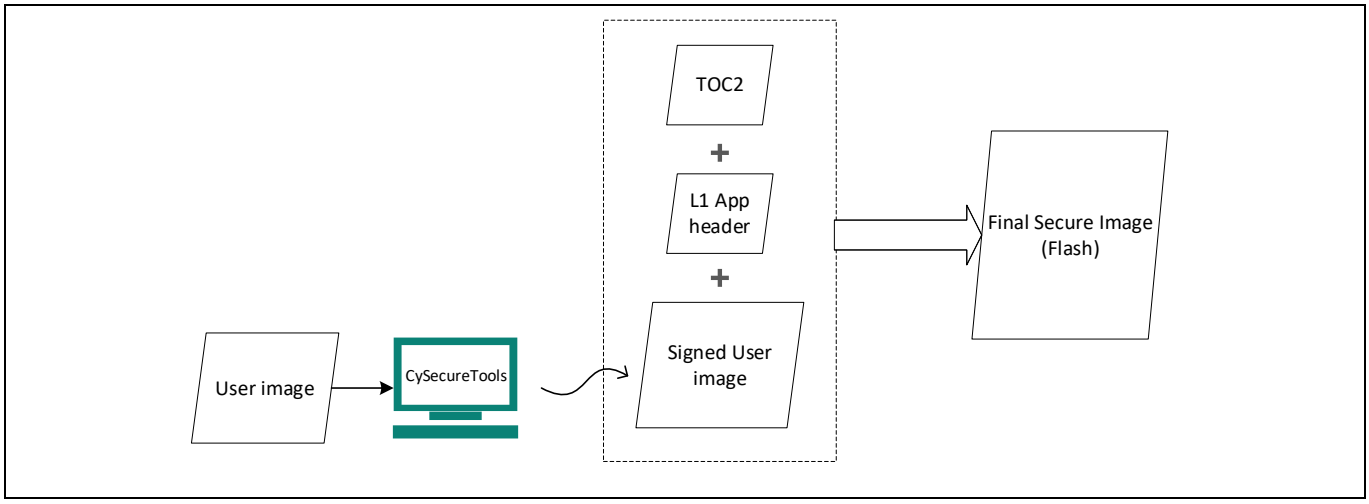

Figure 6 Image combining flow

When using the SECURE lifecycle, there will be an additional header similar to the MCU_Boot header and the corresponding L1App space reserved in the header. Both components need to be combined with a signed user image to create a final binary, which will be programmed into the external flash. Note that the ModusToolbox™ build process combines these files.

5 Implementation on application

This section explains the application execution process and how to use individual commands at each stage. There are two ways to execute the Secure process on CYW20829 silicon. L1 App is signed with RSA2048 key and encrypted with AES-128 bit key (fixed implementation).

1. L2 App is signed with an ECDSA-P256 key and encrypted with random AES-128 (for more details, see the MCUBoot webpage).
2. L2 App is signed with an ECDSA-P256 key and encrypted with the same AES-128 key used to encrypt L1 App.

Do the following steps to while building the basic L1 and L2 app:

a) Generate the keys (signing and encryption) for L1 app and provision the device with same keys.

b) Generate the debug certificate with same keys, which will be used for debugging and programming the device.

c) Use the RSA2048 sign key and AES128 key generated in step a). Build the L1 App (MCUBoot) according to the required method for L2 app and program using the debug certificate.

d) Decide the method for L2 App mentioned above, build the L2 App accordingly and program.

5.1 Tool installation and basics

To perform provisioning and flashing of CYW20829, install the following software and download the dependent prerequisites (All latest version).

- ModusToolbox™
- Python 3.11
- CySecureTools
- MCUBoot

Note: Set the respective PATH variables.

After installation, follow these steps to initialize and edit the device-specific parameters:

- Make sure that the CYW20829 device (dev-kit or custom board) is powered via 2.5 V supply and not regular 3.3 V.
- Install python 3.11 (uninstall any previous python versions).
- Open modus shell (*ModusToolbox\tools_3.2\modus-shell*) and navigate to the desired project folder (create any sample application using ModusToolbox™ if not already).
- Connect the board via KitProg4 or similar device for programming and provisioning the device.
- Install CySecureTools and verify the version by entering the following command in ‘modus shell’:

```
$ cysecuretools –version
```

- In the folder where MCUBoot package is extracted Goto path /cy_mcuboot/boot/cypress
- Initialize the CYW20829 device environment using the following command:

```
$ cysecuretools -t cyw20829 init
```

Implementation on application

- By default, signing is enabled for SECURE LCS. Edit the *policy_secure.json* file in the 'policy' folder, change following values (optionally enable encryption),
  - dead_ap_cm33 = Enable
  - smif_config > encryption = True
- If required, change the key generation path at following place in the *policy_secure.json* file.

```
"pre_build":
    {
        "keys":
        {
            "oem_pub_key_0": {
                "description": "Path to the OEM public key 0",
                "value": "../keys/pub_oem_0.pem"
            },
            "oem_pub_key_1": {
                "description": "Path to the OEM public key 1",
                "value": "../keys/pub_oem_1.pem"
            },
            "encrypt_key": {
                "description": "Path to the AES-128 key for image encryption in
external memory",
                "value": "../keys/encrypt_key.bin"
            }
```

These instructions will initialize the device parameters required for provisioning and key generation.

5.2 Key creation and provisioning

To minimize the risk of using keys from the different private-public pairs for provisioning and image signing, provide a key ID as a command-line option. The commands, which use keys as a parameter still have the option to use the path of the key (The policy contains the necessary information to provision the device). You can generate an Image encryption key by adding a keyword in one single command.

Do the following:

1. Create an OEM key pair by specifying the key ID ('0' or '1'). The OEM private-public key/s will be generated into the files specified in the 'keys' section of the policy. You can create both '0' and '1'.
2. Provision the device.

   Check the policy for the key paths. If the key '0' exists, provision it into the device. If both key '0' and key '1' exist, provision both.

3. Reprovision the device.

Implementation on application

If the key ‘0’ exists, create a reprovisioning packet with the OEM key 0. If the key ‘1’ exists, create a reprovisioning packet with the OEM key ‘0’ and ‘1’. If the key ‘0’ does not exist, it is an error.

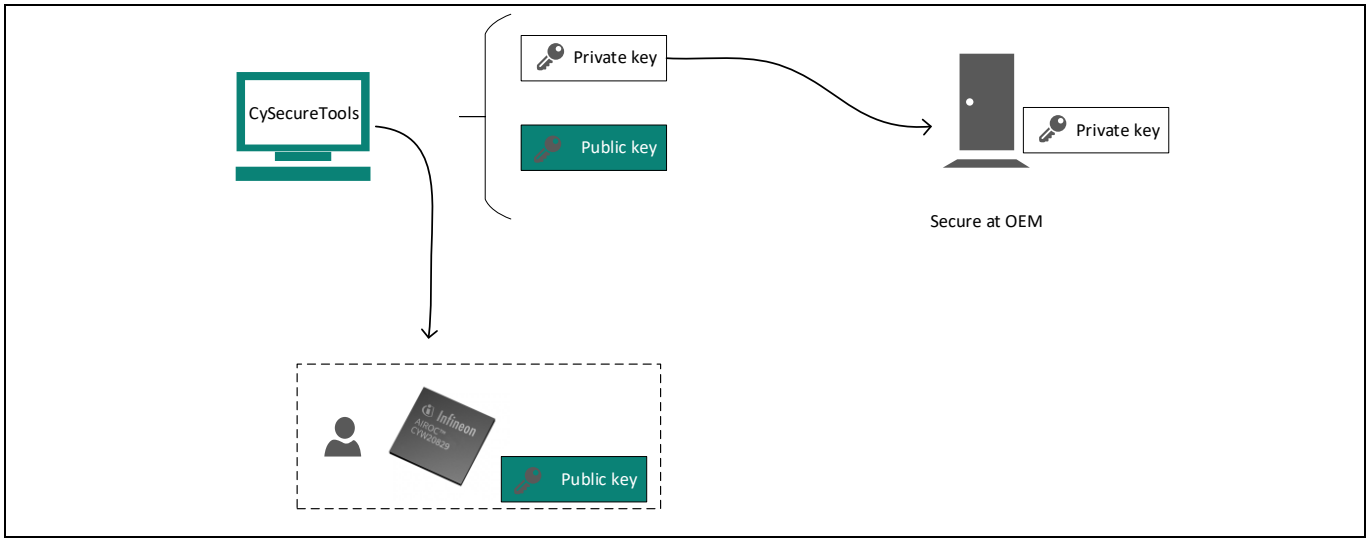

Figure 7 Key creation and distribution

Use the following command to create a signing key for the L1 app (generates a public-private key pair):

```
$ cysecuretools -t cyw20829 -p policy/policy_secure.json create-key --key-id 0
```

Use the following command to create an encryption key for the L1 app:

```
$ cysecuretools -t cyw20829 -p policy/policy_secure.json create-key -aes
```

Create a debug certificate using the same keys with the following command:

```
$ cysecuretools -t cyw20829 -p policy/policy_secure.json debug-certificate --template
packets/debug_cert.json --output packets/debug_cert.bin --key-id 0
```

Provision the device using the following command:

```
$ cysecuretools -t cyw20829 -p policy/policy_secure.json provision-device
```

Note: Note that the provisioning is done only once in the device’s lifetime and so is ‘reprovisioning’. Choose the parameters carefully.

For more details, see “CySecureTools”.

5.3 L1 image building and signing

The bootloader and user application have the signature that is calculated and stored along with the application in the external flash. The application signature is generated by calculating the hash (SHA256) of the application image and generating the signature with the ECDSA NIST P256 algorithm with the OEM private key. The OEM public key is used to verify the signature by BOOT_ROM. You can opt for only a signature or signature along with encrypting the user app as explained in the following section.

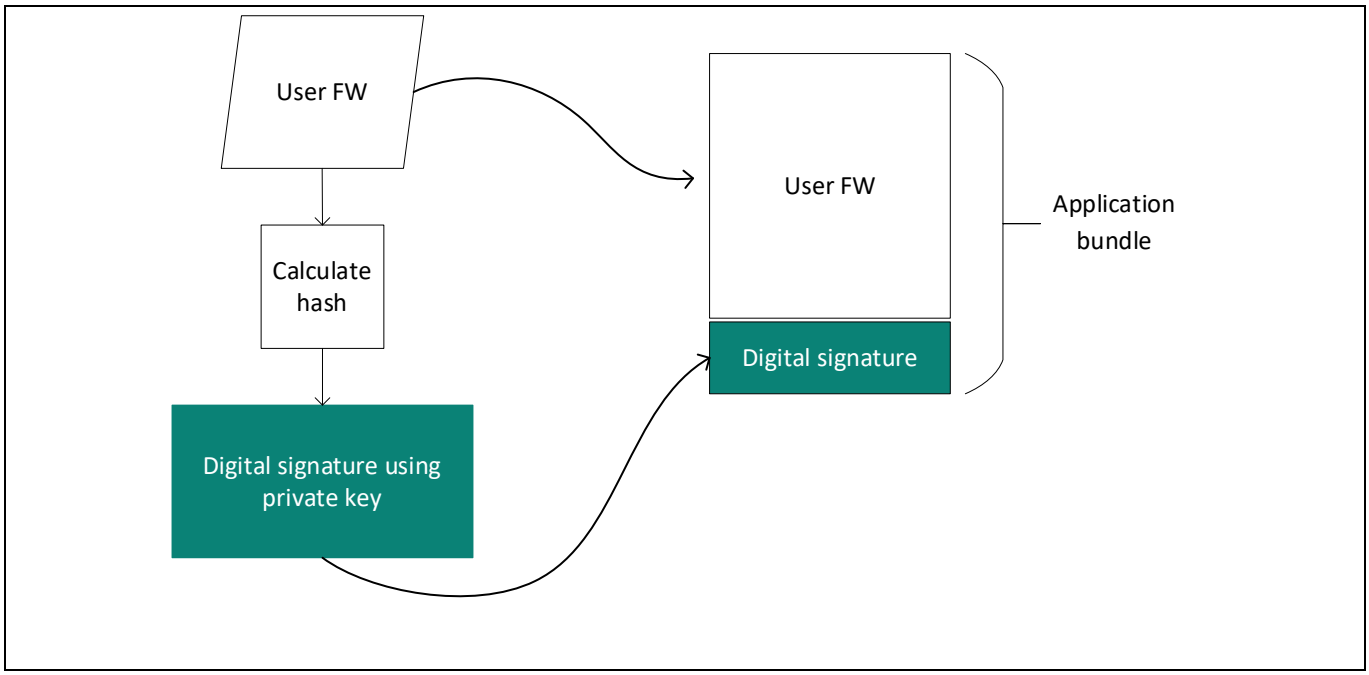

Figure 8 Image signing and appending

Example commands:

Navigate to the *boot\cypress* folder in the MCUBoot folder after applying the required changes in the application code (see the MCUBoot page for implementation),

Execute the following commands to set up tool paths for building the application.

- Run the following command to set the OPENOCD path:

```
export OPENOCD=C:/Users/%user%/ModusToolbox/tools_3.2/openocd
```

- Set toolchain to following:

```
export TOOLCHAIN_PATH=c:/Users/%user%/ModusToolbox/tools_3.2/gcc
```

Build the MCUBoot app using the following command:

For method 1:

```
make clean app APP_NAME=MCUBootApp PLATFORM=CYW20829 BUILDCFG=Debug
FLASH_MAP=platforms/memory/CYW20829/flashmap/cyw20829_xip_swap_single.json LCS=SECURE
ENC_IMG=1
```

The above command will create a signed and encrypted image of the MCUBoot application (LCS=SECURE ENC_IMG=1).

Implementation on application

For method 2:

```
make clean app APP_NAME=MCUBootApp PLATFORM=CYW20829 BUILDCFG=Debug
FLASH_MAP=platforms/memory/CYW20829/flashmap/cyw20829_xip_swap_single.json LCS=SECURE
SMIF_ENC=1
```

The command above will create a signed and encrypted image of MCUBoot application, which will allow booting of L2 application encrypted with the encryption key used to build MCUBoot.

The makefile used in the MCUBoot application folder invokes CySecureTools internally and creates a signed and encrypted image. However, if you want to try manually, use the following command on the desired image:

- Sign image without encryption

```
$ cysecuretools -t cyw20829 -p policy/policy_secure.json sign-image --image image.bin --output image_signed.bin --key-id 0 --image-type APP_PC0
```

- Sign image with encryption

```
$ cysecuretools -t cyw20829 -p policy/policy_secure.json sign-image --image image.bin --output image_signed.bin --key-id 0 --image-type APP_PC0--encrypt --app-addr 0x08000200
```

Generated MCUBoot image is placed in the */MCUBootApp/out/CYW20829/Debug/MCUBootApp.hex* file.

Flash this image using the following steps:

- Erase the existing Flash content using the following command:

  ```
  ${OPENOCD}/bin/openocd -s "$OPENOCD/scripts" -f
  "$OPENOCD/scripts/interface/kitprog3.cfg" -c "set ENABLE_ACQUIRE 0" -c "set
  DEBUG_CERTIFICATE ./packets/debug_cert.bin" -c "set SMIF_BANKS { 0 {addr
  0x60000000 size 0x800000 psize 0x100 esize 0x40000} }" -f
  $OPENOCD/scripts/target/cyw20829.cfg -c "init; reset init; flash erase_address
  0x60000000 0x100000; shutdown"
  ```

- Program the built MCUBoot app to device using the following command:

  ```
  ${OPENOCD}/bin/openocd -s "$OPENOCD/scripts" -f
  "$OPENOCD/scripts/interface/kitprog3.cfg" -c "set ENABLE_ACQUIRE 0" -c "set
  DEBUG_CERTIFICATE ./packets/debug_cert.bin" -c "set SMIF_BANKS { 0 {addr
  0x60000000 size 0x100000 psize 0x100 esize 0x1000} }" -f
  $OPENOCD/scripts/target/cyw20829.cfg -c "init; reset init; flash write_image erase
  "./MCUBootApp/out/CYW20829/Debug/MCUBootApp.hex" 0x00000000; init; reset init;
  reset run; shutdown"
  ```

Note: *Note that erasing external memory is required when flashing a new application.*

5.4 L2 (sample) app guidelines

Make sure that the device is provisioned in SECURE LCS according to the Key creation and provisioning section. The sample application is provided in the MCUBoot folder, and you can verify the chain of trust by building and flashing the same. Note that the MCUBoot application (L1 app) must be flashed before executing the following commands for the L2 app:

1. Navigate to the *boot\cypress* folder in the MCUBoot folder.
2. For method 1, use the following command:

   ```
   make clean app APP_NAME=BlinkyApp IMG_TYPE=BOOT PLATFORM=CYW20829 BUILDCFG=Debug
   FLASH_MAP=platforms/memory/CYW20829/flashmap/cyw20829_xip_swap_single.json
   ENC_IMG=1
   ```

   Flash the BlinkyApp using the following command:

Implementation on application

```
${OPENOCD}/bin/openocd -s "$OPENOCD/scripts" -f
"$OPENOCD/scripts/interface/kitprog3.cfg" -c "set ENABLE_ACQUIRE 0" -c "set
DEBUG_CERTIFICATE ./packets/debug_cert.bin" -c "set SMIF_BANKS { 0 {addr
0x60000000 size 0x100000 psize 0x100 esize 0x1000} }" -f
$OPENOCD/scripts/target/cyw20829.cfg -c "init; reset init; flash write_image
"./BlinkyApp/out/CYW20829/Debug/boot/BlinkyApp.bin" 0x60020000; init; reset init;
reset run; shutdown"
```

3. For method 2, use the following command to build a BlinkyApp application:

```
make clean app APP_NAME=BlinkyApp IMG_TYPE=BOOT PLATFORM=CYW20829 BUILDCFG=Debug
FLASH_MAP=platforms/memory/CYW20829/flashmap/cyw20829_xip_swap_single.json
SMIF_ENC=1
```

4. For method 2, use the following command to encrypt the BlinkyApp binary:

```
cysecuretools -t cyw20829 encrypt --input
./BlinkyApp/out/CYW20829/Debug/boot/BlinkyApp.bin --output
./BlinkyApp/out/CYW20829/Debug/boot/BlinkyApp_encrypted.bin --iv 0x08020000 --
enckey keys/encrypt_key.bin --nonce
./MCUBootApp/out/CYW20829/Debug/MCUBootApp.signed_nonce.bin
```

5. Flash the application using the following command for method 2:

```
${OPENOCD}/bin/openocd -s "$OPENOCD/scripts" -f
"$OPENOCD/scripts/interface/kitprog3.cfg" -c "set ENABLE_ACQUIRE 0" -c "set
DEBUG_CERTIFICATE ./packets/debug_cert.bin" -c "set SMIF_BANKS { 0 {addr
0x60000000 size 0x100000 psize 0x100 esize 0x1000} }" -f
$OPENOCD/scripts/target/cyw20829.cfg -c "init; reset init; flash write_image
"./BlinkyApp/out/CYW20829/Debug/boot/BlinkyApp_encrypted.bin" 0x60020000; init;
reset init; reset run; shutdown".
```

5.5 Reference code example

See an existing code example in GitHub for secure silicon implementation. Go to the mtb-example-btstack-freertos-cyw20829-battery-server-ota GitHub page and read README.md to understand the working of it. Code Example includes OTA, which helps transferring upgraded firmware to CYW20829. In the provided code example, the following points are covered:

- How to clone and build MCUBoot with techniques mentioned in the Implementation on application section.
- How to install CySecureTools, which will be used in generating signed and encrypted images internally by ModusToolbox™.
- How to build and flash the user application in both SECURE and NO_SECURE mode.
- How to build upgrade image for existing application.
- How to use Bluetooth® utility to transfer upgrade image to CYW20829.

6 Summary

This guide explored the CYW20829 MCU device security architecture, the associated development tools, and the steps to configure a secure environment for CYW20829 using “CySecureTools” and ModusToolbox™. CYW20829 is a Bluetooth® 5.4-compliant, standalone baseband processor with an integrated 2.4-GHz transceiver with support for Bluetooth® Low Energy. The device is intended to use in audio (source), sensors (medical, home, and security), HID, and remote-control functionalities as well as a host for other IoT applications. Configuring CYW20829 for a secure environment can enhance its usability in applications where overall security is a prime factor.

References

[1] Device documentation

- CYW20829 MCU datasheet

[2] Development kits

- CYW920829M2EVK-02
- CYW920829B0M2P4TAI100-EVK
- CYW920829B0M2P4EPI100-EVK

[3] Code examples

- Code examples for ModusToolbox™ software: Visit this code example for a comprehensive collection of code examples using the ModusToolbox™ IDE.

[4] Tool documentation

- ModusToolbox™ software user guide

[5] Tool

- ModusToolbox™ software: The Infineon IDE for IoT designers
- "CySecureTool": Command-based tool, which is used for provisioning.

Revision history

Document revision	Date	Description of changes
**	2024-03-05	Initial release.
*A	2024-08-27	Restructured Overview of implementation and Implementation on application: Added detailed explanation and implementation using actual commands, which can be executed directly on CYW20829. Updated Reference code example: Added existing code example link and described the functionality.

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.
The Bluetooth® word mark and logos are registered trademarks owned by Bluetooth SIG, Inc., and any use of such marks by Infineon is under license.

Edition 2024-08-27
Published by

Infineon Technologies AG
81726 Munich, Germany

Do you have a question about this document?

Email:

erratum@infineon.com

Document reference

002-39500 Rev. *A

Important notice
The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics (“Beschaffenheitsgarantie”)

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer’s compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer’s products and any use of the product of Infineon Technologies in customer’s applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer’s technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Warnings
Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies’ products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.