Security Comparison Between PSoC® 64 Secure MCU and PSoC 62/63 MCU Whitepaper

Erik Wood and Sree Harsha Angara – Cypress Semiconductor Corp.

Introduction

Security is a “must-have” for embedded IoT designs. But not all security is the same. There are “good,” “better,” and “best” levels of protection, and features like ease-of-use and time-to-market in the design cycle must also be considered. When it comes to IoT security, “credibility” is won by faithfully executing industry best practices. Fundamental choices in hardware and software can make or break a secure IoT design. To aid designers in making these choices, Arm® has introduced its Platform Security Architecture (PSA), an MCU security architecture supported by Trusted Firmware – M (TF-M).

Cypress’ PSoC® 64 family of microcontrollers is one of the first to support PSA and deliver TF-M. The PSoC 64 Secure MCU provides for three-levels of hardware- and firmware-based resource isolation, which is the highest level of isolation defined by the PSA. The Arm Dual-Cortex® -M-core, system-on-chip offers a secure M0+ core, physically separated from a user application running on the other M4 core. In addition, the PSoC 64 Secure MCU offers secure element functionality that can be used to build and authenticate secure applications. PSoC 64 Secure MCUs provide a root-of-trust and a mechanism to transfer that root-of-trust to customers and OEMs during the provisioning process while also supporting a chain-of-trust (CoT). This chain-of-trust also helps establish the on-chip chain-of-trust that protects applications from being altered by calculating and storing hashes over code blocks and signing these with private/public key pairs.

Perhaps the most-crucial point of the PSoC 64 Secure MCU is that all offered security features come ready-to-use out-of-the-box. This is vital to supporting the realization that one trillion IoT devices will be securely deployed by 2025, where a significant percentage of these devices will be developed by small- to medium sized teams that may not have the resources to support security firmware development and instead need on-chip security within the MCU.

PSoC 64 Secure MCU and PSoC 62/63 MCU Summary Comparison Table

This table is an easy way to quickly understand the distinguishing features of the PSoC 64 Secure MCU

#	What?	PSoC64	PSoC62
1	Secure Boot	Out-of-box support with ECC P-256 keys	Out-of-box support with RSA2048 keys
2	Securely update firmware	Pre-built Cypress secure bootloader binary	Secure image code base, bootloader SDK and code examples as templates
3	Runtime protection, isolate environments	Out-of-box protection context setup and TF-M support	Secure image code base as a template for protection context setup
4	Protect data at rest	All debug access ports can be shut down by setting up provisioning policies	All debug access ports can be shut down by setting up registers
5	Protect data in transit	Cryptographic accelerators, mbedTLS software library support	Cryptographic accelerators, mbedTLS software library support
6	Securely manufacture your product	Provisioning scheme allows transfer of root-of-trust, forming unique device identity and certificate	Do-it-Yourself (DIY)
7	Certifications	PSA Level-1 PSA Level-2* FIPS 140-2 Level 1*	Do-it-Yourself (DIY)

* Available in Q2, 2020

Architectural Security Requirements for an IoT device

PSoC 62/63 MCUs can be used to make secure IoT products. These MCUs offer a considerable degree of flexibility regarding how to implement do it yourself (DIY) secure firmware to leverage the existing hardware-based security and crypto drivers. However, a designer must consider a couple of factors when using PSoC 62/63 MCU as a secure solution: 1) Their team is qualified for the DIY secure firmware development and deployment process, and 2) They trust the supply chain that brought their PSoC 62/63 MCU to their design and production environments. Time-to-market, development cycles, and security pre-certification should also be noted when considering the DIY approach.

Consider a use-case where you need to design and manufacture an IoT sensor node. Typically, you would start by thinking about your design from a threat point of view. The Threat-based Analysis Method for IoT Devices paper gives an excellent overview of this process. Without going into a detailed PSA-specific analysis, the general set of features developers need to consider are as follows:

#	What?	Why?	How?
1	Secure Boot	Guarantees that only your authorized firmware is being executed	Cryptographic signature check on firmware by boot code before launch
2	Securely update your firmware	Avoid malicious code from taking over your device	Secure FOTA updates using cryptographically signed firmware
3	Runtime protection, isolate secure and non-secure environments	If your product becomes vulnerable to a future exploit, isolate resource access to limit amount of damage that can be done	Define regions of protection for each execution environment and dedicate resources and memory regions
4	Protect data at rest: restrict physical access to firmware and sensitive data	Firmware images can be reverse-engineered to expose weaknesses	Close out any debug ports, restrict access to sensitive information. Encrypt any data being stored external to the chip. Secure key storage for any private keys stored on chip.
5	Protect Data-in-Transit: Securely transmit data to your cloud server	Data privacy, user credentials being presented must be protected	Cryptography, typically TLS-based
6	Securely manufacture your product	Prevent your design from being easily replicated by contract manufacturers	Requires an auditable manufacturing trail from assembly line to cloud to ensure no devices are spoofed/duplicated
7	Certifications	Some end-markets require security certificates for regulatory reasons.	Reach out to accredited labs and test your final product

While PSoC 62/63 and PSoC 64 provide a way to do all the above, PSoC 64 provides out-of-box functionality to provide a shorter time-to-market.

Design Example – Smart Lock

The best way to highlight the differences between the PSoC 62/63 and the PSoC 64 line of MCUs can be done by running through a simple design example. Consider a case where you want to make a smart lock that connects to the internet and sends lock/unlock instructions through either Wi-Fi® or BLE.

From a functional perspective, the typical flow of firmware you would have is shown in Figure 1.

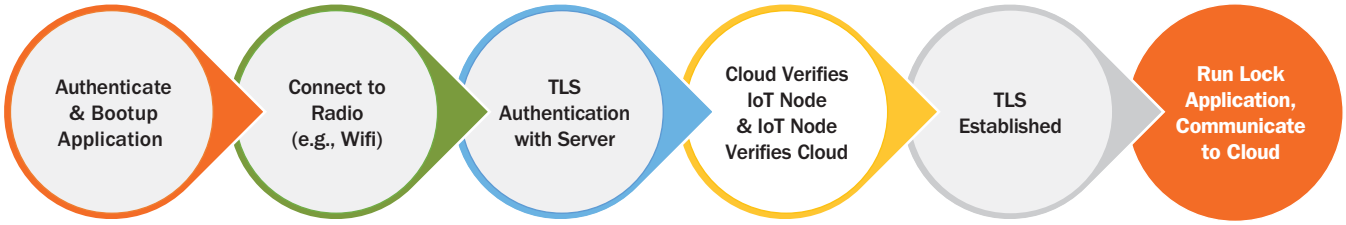

Figure 1: Smartlock firmware flow

1) On bootup, the device first authenticates the code which must be executed. This is typically done by verifying the signature of the application image against a known public key provisioned in the device.
2) After successful bootup, the device will interface to a radio, like Wi-Fi/BLE, and connect to a local wireless gateway.
3) Now, the device will need to initiate a connection to the cloud service. The cloud-side server typically will log data, do some front-end data visualization, and perform other functions. To successfully connect to the cloud server, two fundamental things must happen:
    a) The IoT node must trust the cloud server
    This is done by normal server-side TLS which is the way modern browsers verify identity. A server certificate is present in the device (put in during device programming) which it uses to do a 'Hello' and challenge/response to validate that the server is authentic.
    b) The cloud server must trust the IoT node
    For IoT sensor applications, there is an additional step where the cloud must know who is trying to talk to it. This client-side authentication is always a requirement as you don't want unknown devices to abuse your cloud infrastructure. To achieve this, a unique, trusted certificate is put into every device during the manufacturing process. The server will validate the certificate presented by the device, perform a challenge/response to prove private key ownership.
4) At this point, TLS can be established with a shared secret being formed and the data to and from the server is fully encrypted.
5) Finally, the application runs, processes any commands, and sends/receives from cloud.

Product Manufacturing Requirements

Step 3(b) in the above scheme is a critical manufacturing step, where a unique, trusted certificate must be put into every device. There are two ways you can go about this:

Option 1: Generate unique certificates from the cloud service, provision your devices with the certificates

You, the OEM, would generate a batch of certificates you trust; equal to the number of devices you would like to manufacture as shown in Figure 2.

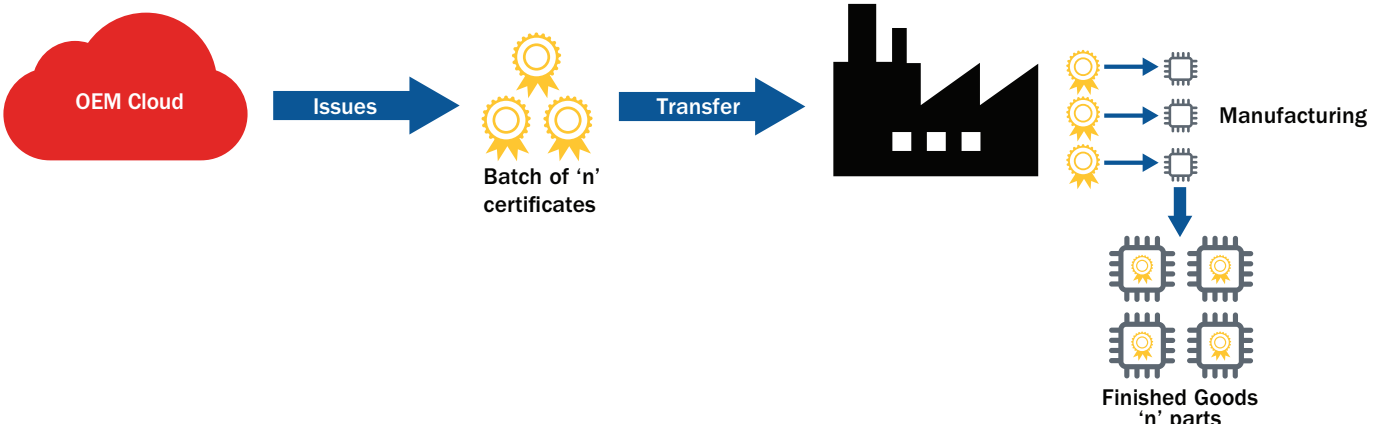

Figure 2: Generate device credentials on cloud

This approach works well for most applications. However, there is subtlety which must be noted here. The device certificate contents do not have any connection with the unique device ID or any other security information (e.g., chip public key). In addition, you will need to issue new certificates and go through the certificate transfer process anytime you want to make a new batch of devices.

Option 2: Just-in-time provisioning and on-boarding

The other way to generate a unique certificate is to use the hardware root-of-trust (which includes a unique ID and a random device key pair) with a trusted intermediate manufacturing environment to generate device certificates on the manufacturing line. This eases the logistical burden and allows flexibility for making more devices much more easily once the manufacturing line is already setup as shown in Figure 3.

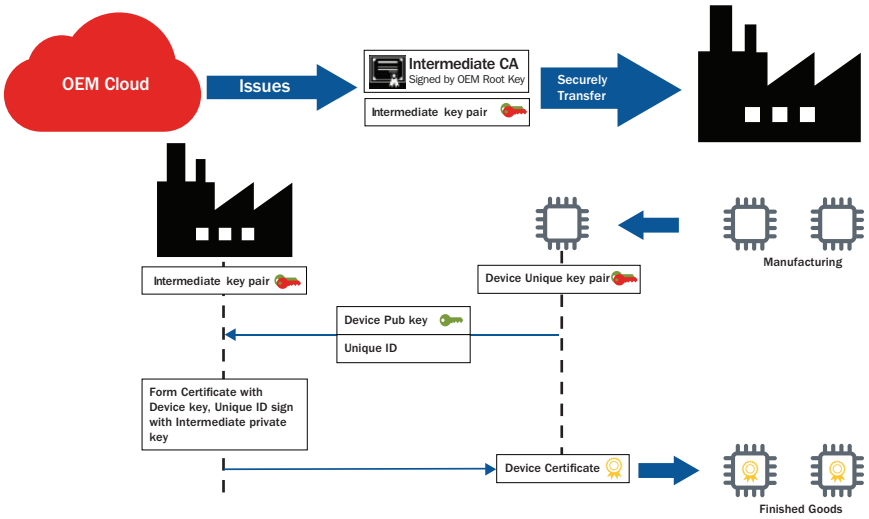

Figure 3: Generate device credentials at secure manufacturing facility

The typical onboarding flow using this type of device certificate is shown in Figure 4.

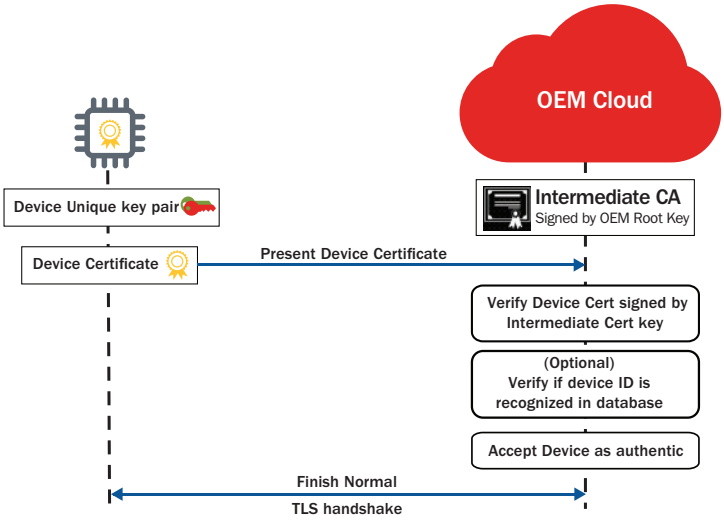

Figure 4: Device onboarding flow using generated device credentials

Boot-time Security Requirements - Secure Boot

Secure Boot is a fundamental requirement in our smart lock application. This feature ensures that only your firmware is authorized to run on the chip. Arm® PSA specifies Secure Boot launch as a series of image launches, based on an immutable, hardware-based, root-of-trust. Verification of each stage is based on a cryptographic signature, which ties the ownership of the code to the developer/owner of the private key as shown in Figure 5.

Figure 5: Boot time security requirements - Secure Boot flow

Runtime Security Requirements

In addition to the Secure Boot, you will need to setup isolation boundaries between the secure and non-secure world to restrict access to sensitive material in case of any future vulnerabilities exposed. On a high level, if you had to architect the various pieces of firmware you would end up with something like what is shown in Figure 6.

The split of functions would be:

Immutable Hardware Root-of-Trust and Services:

1) Has Secure Boot code that is responsible for launching and updating applications on bootup
2) Any device root keys and certificates

Secure Operating Environment:

1) TF-M library, includes a cryptography library, which can call into an immutable root-of-trust when needed
2) Secure services like user credential management (e.g., register a new user) and the lock application

Non-Secure Operating Environment:

1) All rich application functions, RTOS
2) Wireless stacks, drivers (WHD), cloud middleware and Peripheral Driver Libraries (PDL)

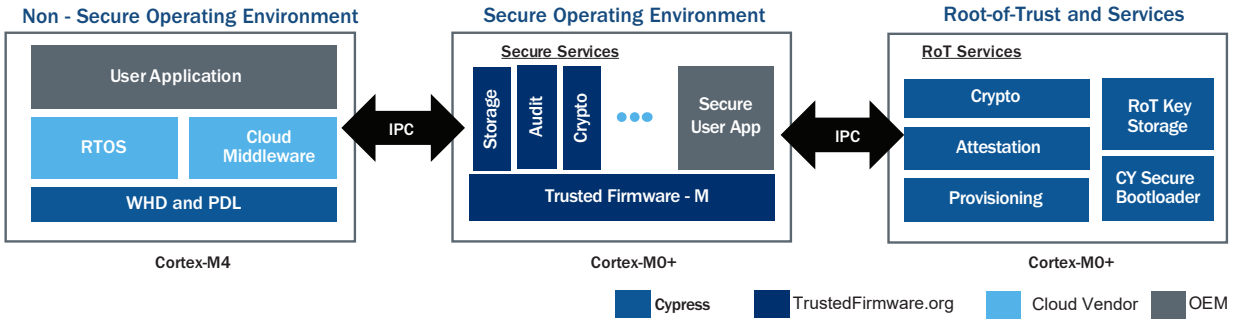

Figure 6: Runtime security requirements

Implementing the Design – PSoC 62 vs PSoC 64

Now that we have listed out the high-level requirements of our product, let's look at how we would go about implementing such a design.

Secure Boot:

The PSA approach defines that an MCU must start up in a trusted manner based on the root-of-trust. In both PSoC 62 and PSoC 64, this is ensured by a section of ROM code that will calculate a hash over the next section of flash executable code and compare that hash with one that is stored in a write-once memory location (a hardware register called eFUSE). The processor will start executing code from the next section only if the hashes match. This immutable trusted root is the first building block used by subsequent applications. This code then proceeds to verify the next (mutable) image, by verifying the integrity and authenticity of the signature against a public key provisioned in the part.

Implementing a Secure Boot design for both PSoC 62 and PSoC 64 fundamentally require the same series of steps:

1) Generate an OEM RoT key pair; typically, one key-pair for a version of the product
2) Program the OEM RoT public key into the device
3) Program an image which is signed by the OEM RoT private key (or a chain-of-trust key based on this root key)

However, there is a subtlety associated with how you manufacture your devices when it comes to implementing this feature. The fundamental difference comes in how the OEM root public key pair gets injected into the device.

For a PSoC 62 device, you would hand over your public key, as well as a signed image, over to your Contract Manufacturer (CM) and ask them to program both. In this process, there is no cryptographic basis by which you can check that your key got into the device. It is essentially "uncontrolled" where you trust your CM not to change this key out and put a root-kit before launching your image. This is not a problem if the CM is trusted and specifies a trust/key hand-off process outside of the device scope.

When using PSoC 64, the OEM RoT public key can only by injected into the device by following a series of cryptographic handshakes between the device and the programming equipment to hand over the root-of-trust from Cypress to the OEM.

This is achieved by injecting a Cypress root public key into every PSoC 64 part. For a device to be provisioned with the OEM RoT public key, the manufacturing programmer (HSM) must be trusted by both Cypress and the OEM. The exact process is beyond the scope of this paper, but a high-level description is present in the secure manufacturing section. The manufacturing process differences between PSoC 62 and PSoC 64 is shown in Figure 7.

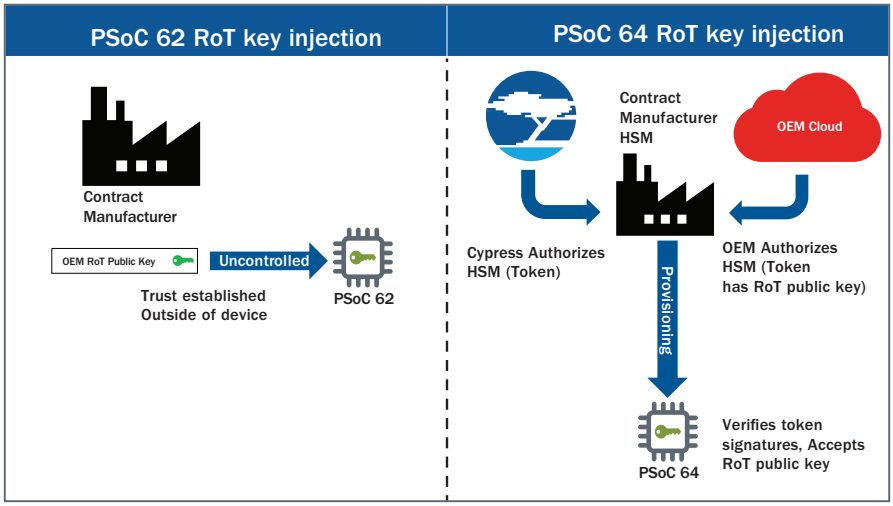

Figure 7: PSoC 62 vs PSoC 64 Root-of-Trust injection

PSoC 64 enables this hand over of the root-of-trust to the OEM RoT key as a one-time, cryptographically binding process which leaves an auditable trail in the chip. The PSoC 64 provisioning process enables a ready-to-use framework with secure manufacturing partners to help accelerate time-to-market.

Secure Update:

The natural next step to Secure Boot is secure update, where you want to update the image and sign it with the RoT public key present in the device.

When using the PSoC 62 device, a secure image code base, a bootloader SDK and code examples are provided to be used as a template to build your own Secure Update mechanism.

When using a PSoC 64 device, a pre-built Cypress Secure Bootloader will be made available. This is a pre-built image that enables:

- Open-source MCU boot implementation
- Protection contexts for runtime protection of bootloader code and key material
- Ability to use keys provisioned in the device during secure manufacturing

PSoC 64 simplifies implementing secure upgrades by having the ability to setup image-signing keys, image addresses, and sizes as a simple JSON-editable file. This JSON file, called a “Boot & Upgrade” policy, will be signed by the OEM RoT private key and injected during the secure manufacturing phase.

The Cypress Secure Bootloader uses public keys provisioned into the device during secure manufacturing to validate the next stage or an update in the staging area before updating the running image. See Figure 8 for a high-level overview of the differences.

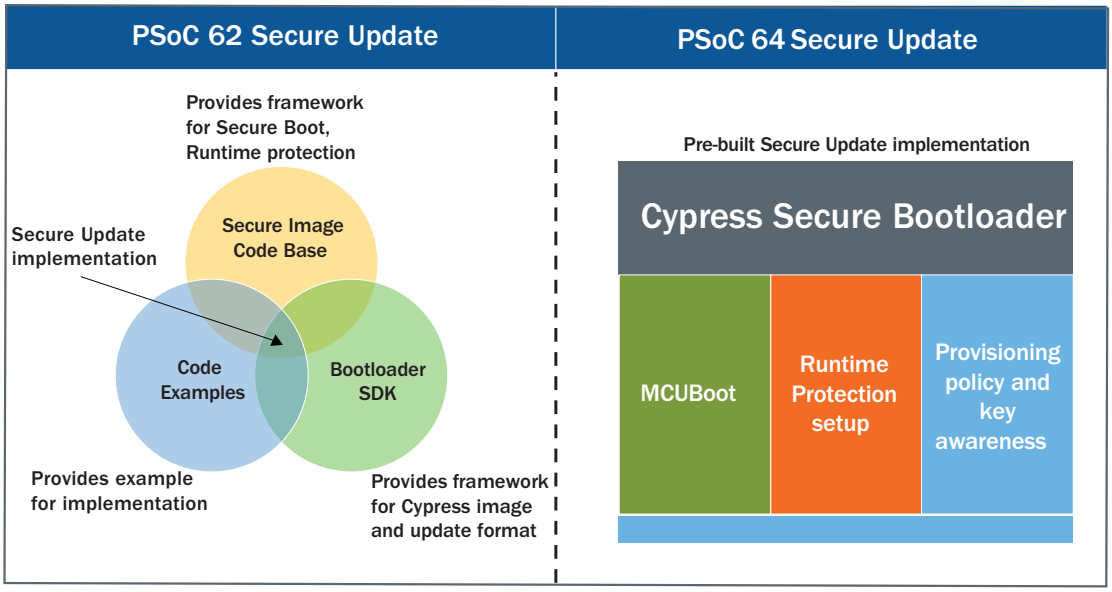

Figure 8: PSoC 62 vs. PSoC 64 Secure Update

Runtime protection, Resource Isolation:

Both PSoC 62 and PSoC 64 enable runtime protection (resource isolation) by using the same Shared Memory Protection Units (SMPU) and Peripheral Protection Units (PPU) hardware peripherals. These hardware peripherals allow you to setup protection contexts (PC) around regions of Flash/RAM and specify which bus masters (CM0+/CM4/DMA) are allowed to access memory.

When using PSoC 62, example implementations of SMPU and PPU setup can be found in the secure image code base. These settings may require some modifications depending on your application.

In PSoC 64 the SMPU and PPU settings are automatically setup by the immutable boot code and Cypress Secure Bootloader. In addition, PSoC 64 enables Trusted Firmware-M (TF-M) for Arm-v7M. This provides an intuitive framework to add secure services such as key storage, cryptography, etc., as well as adding your custom services like user registration. See Figure 9 for the various protection contexts setup on a PSoC 64.

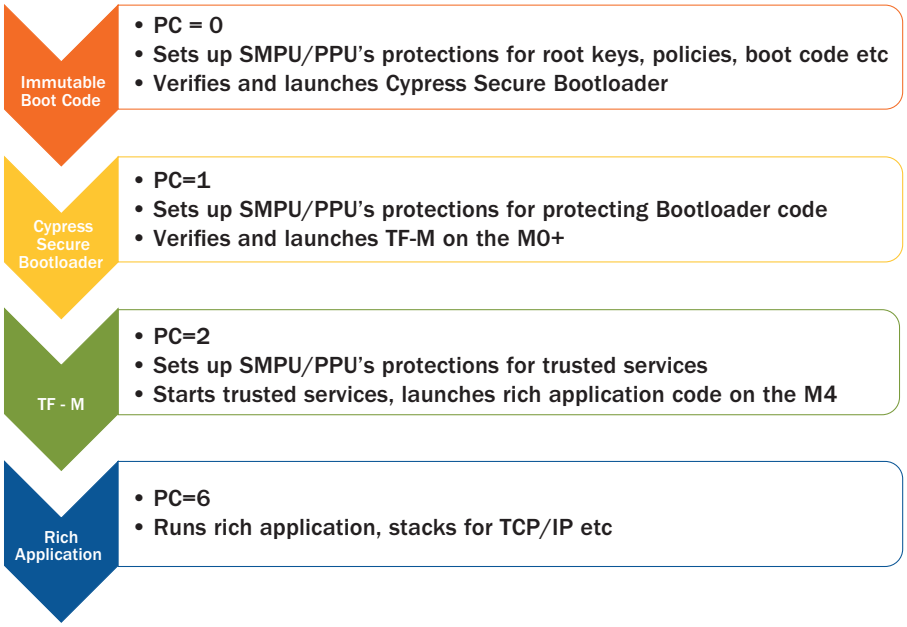

Figure 9: Resource isolation with PSoC 64 protection contexts

Protecting Data-at-Rest and Transit:

Both PSoC 62 and PSoC 64 provide mechanisms to close out debug ports individually for each core (CM0+, CM4), as well as the SysAP port during manufacturing. In PSoC 62, this is provided as a setup option in the secure image code base, as well as a manufacturing step where the device settings are made immutable by blowing eFuses.

PSoC 64 takes advantage of the provisioning scheme and achieves the same functionality by providing a user-editable JSON file. This JSON file, called a debug policy, will be signed by the OEM RoT private key and injected during the secure manufacturing phase. The settings are made immutable by blowing eFuses as well.

Both PSoC 62 and PSoC 64 offer cryptographic accelerators for a variety of symmetric, asymmetric, and hashing operations. Both support popular libraries, such as mbedTLS, to allow for protecting data-in-transit.

Secure Manufacturing:

PSoC 62 does not offer any out-of-box features that enable just-in-time provisioning or unique device certificate generation. If this is required, you must write the relevant code for both the manufacturing line and the PSoC 62 firmware.

PSoC 64 is purpose-built for this use case and the provisioning process not only allows a root-of-trust transfer, but it also exposes several features that uniquely lend themselves to manufacturing secure IoT devices. The process of provisioning the chip can be thought of as two steps:

1) Taking over the root-of-trust

2) Injecting any OEM-specific assets like keys, security policies, and certificates

Taking over Root-of-Trust with PSoC 64

The root-of-trust takeover process can be thought of as a series of trust claims; exchanged between four entities:

a) Cypress – Holder of the Cypress root private key

b) Hardware Security Module (HSM) – The entity which is authorized to provision and program the PSoC 64

c) OEM/User – The effective user/code developer of the part

d) PSoC 64 device – Holder of the Cypress root public key

Prior to running the root-of-trust transfer procedure, a procedure called the 'entrance exam' can be run on the chip which will verify that no malicious firmware is already running on the device. The entrance exam runs a hash over the internal flash of the device and compares it against an expected hash value. If there is a mismatch, the HSM can detect firmware on the device and erase the chip before proceeding further. See Figure 10 for a visual representation of the flow.

The series of steps which take place are:

1) Cypress authorizes the HSM to allow it to provision the 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 device

4) The PSoC 64 verifies authorization signatures and verifies claims. If okay, it accepts the OEM key and allows the HSM to further send provisioning packets

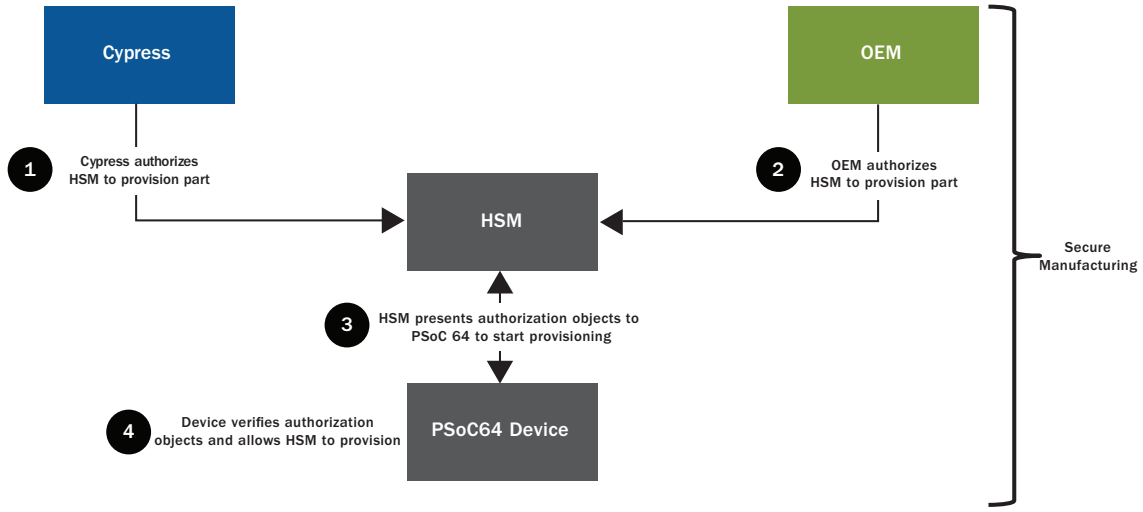

Figure 10: Root-of-Trust take-over flow

Injecting OEM-Specific Assets with PSoC 64

See Figure 11 for a visual representation of the flow.

1) In the first step, the root-of-trust packet is sent to the PSoC 64

2) When the root-of-trust token is verified and accepted, a unique key pair is generated on the chip and the public key is exported along with the unique ID and OEM RoT public key as a self-signed token.

3) The HSM can verify the signature on the device response packet to ensure that the key is valid, it then signs this response packet with the intermediate key trusted by the OEM (as defined in the just-in-time provisioning) and forms the device certificate

4) The HSM then sends any other OEM assets, like the device policies and other OEM keys, to the PSoC 64

5) After the device verifies the signatures of incoming packets, it accepts and enforces the policies, keys and certificates to finalize the part

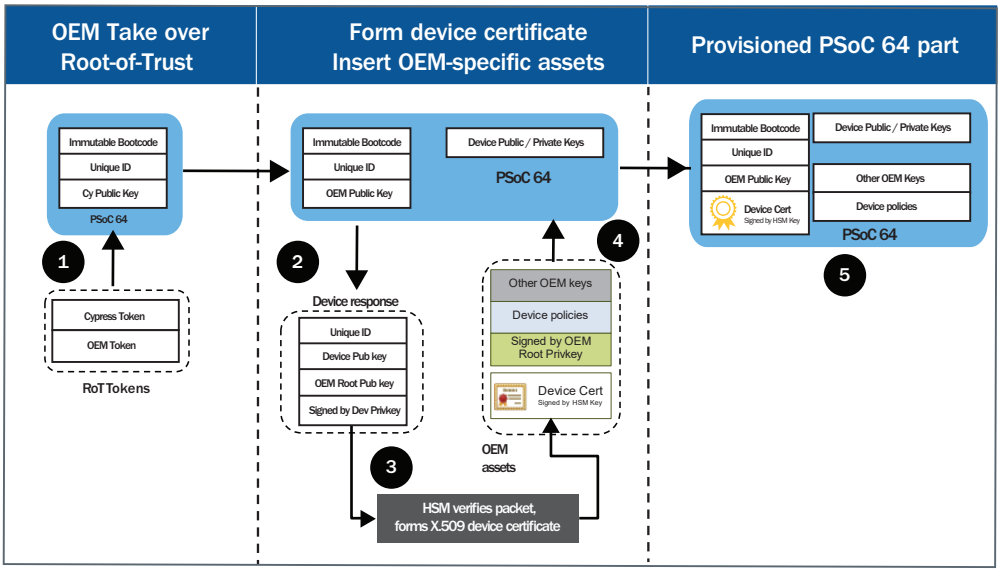

Figure 11: Creating device certificate and injecting OEM-specific assets

It is important to note that although only assets such as keys and policies are covered in the flow diagram, firmware images, such as Cypress Secure Bootloader and OEM applications, are generally programmed in the same step where OEM-specific assets are injected. This programming step is no different from the standard flash programming interface provided across all PSoC 6 products.

This provisioning framework allows PSoC 64 to follow a standardized series of cryptographically auditable steps. In addition, Cypress enables trusted programming partners with easy deployment of secure systems using the PSoC 64 family. From a security perspective, this reduces the developer's burden to forming provisioning tokens populated with the public keys and policies required for their application and sending these provisioning tokens, as well as their developed code, to a trusted programming partner.

PSoC 64 Secure MCU Certifications

Beyond the added security features and out-of-the-box convenience and assuredness of "just works" IoT MCU security, Cypress will be uniquely and exclusively investing in security certifications for PSoC 64 and not PSoC 62. A set of unique PSoC 64 MPNs will support firmware changes that qualify them for FIPS 140-2 Level 1 certification and PSA Level 1 and Level 2 certification.

FIPS 140-2 is a NIST (U.S. Government) standard that attests the hardware and firmware implementation to support cryptographic operations, key storage, and random-number-generator entropy to meet a baseline security threshold. PSA Level 1 is a relatively new standard run by PSA.org to certify that an MCU supports a quality hardware-based root-of-trust. PSA Level 2 builds on Level 1 by also certifying that TF-M has been implemented properly and that the MCU supports reasonable countermeasures against low- to medium-cost fault injection and side-channel attacks.

Having a pre-certified MCU not only supports a mutual understanding that the MCU meets or exceeds an industry benchmark, it also supports the likelihood that an IoT device made with a certified MCU will also achieve the same certification. This reduces time and cost for the end-product certification process itself and adds tremendous value as FIPS 140-2 and PSA L1 and L2 are becoming more common in the world of IoT devices.

Benefits of Open-Source Firmware

PSA is supported by an open-source firmware stack, developed by Arm, called TF-M. It is built on a common foundation and is meant to reduce costs, accelerate time-to-market, and enable users and consumers to trust their IoT device security. The TF-M stack is governed by TF-M.org and forms the essential services for the Secure Processing Environment and manages the isolated secure M0+ core in the PSoC 64 Secure MCU which is further aided by the cryptographic hardware acceleration built into the chip to achieve power-efficient and high-speed performance.

Conclusion

Security is a foundational feature required for mass adoption of IoT solutions. On-chip hardware and software security, like what is offered with Arm® PSA and Cypress' PSoC 64 Secure MCUs, is required to establish a root-of-trust that can then be used to authenticate each subsequent application, no matter who creates it or programs it. PSoC 62/63 MCUs are industry-leading IoT MCUs that offers a platform where a team of security firmware developers can deploy DIY code to create a secure solution. PSoC 64 Secure MCUs builds on those capabilities to provide out-of-the-box secure MCU solutions based on industry best practices with fully supported and optimized TF-M firmware from Arm.

Further reading:

1. https://pages.arm.com/PSA-Building-a-secure-IoT.html
2. http://www.cypress.com/documentation/application-notes/an221111-psoc-6-mcu-creating-secure-system
3. http://www.cypress.com/products/modustoolbox-integrated-design-environment-ide

About PSoC 6

PSoC 6 is the industry's lowest power, most-flexible MCU architecture with built-in Bluetooth Low Energy wireless connectivity and integrated hardware-based security in a single device. Software-defined peripherals can be used to create custom analog front-ends (AFEs) or digital interfaces for innovative system components such as electronic-ink displays. The architecture offers flexible wireless connectivity options, including fully integrated Bluetooth Low Energy (BLE) 5.0. The PSoC 6 MCU architecture features the latest generation of Cypress' industry-leading CapSense® capacitive-sensing technology, enabling modern touch and gesture-based interfaces that are robust and reliable. The architecture is supported by Cypress' PSoC Creator™ Integrated Design Environment (IDE) and the expansive Arm ecosystem. Designers can find more information on the PSoC 6 MCU architecture at http://www.cypress.com/PSoC6.

About Cypress

Cypress is the leader in advanced embedded system solutions for the world's most-innovative automotive and industrial applications, smart-home appliances, consumer electronics and medical products. Cypress' microcontrollers, analog ICs, wireless and USB-based connectivity solutions and reliable, high-performance memories help engineers design differentiated products and get them to market first. Cypress is committed to providing customers with the best support and development resources on the planet enabling them to disrupt markets by creating new product categories in record time. To learn more, please visit us online at www.cypress.com.

Cypress Semiconductor Corporation
198 Champion Court, San Jose CA 95134
phone +1 408.943.2600 fax +1 408.943.6848
toll free +1 800.858.1810 (U.S. only) Press "1" to reach your local sales representative

002-28236**