J. Beningo, *Embedded Software Design*
https://doi.org/10.1007/978-1-4842-8279-3_3

# 3. Secure Application Design

Jacob Beningo[1]
(1) Linden, MI, USA

Every embedded system requires some level of security. Yet, security is often the most overlooked aspect in most software designs. Developers may look at their product and say it's not an IoT device, so there is no need to add security. However, they overlook that the same firmware they deploy in their systems may be worth millions of dollars. A competitor interested in reverse-engineering the product or a hacker interested in tampering with the device locally is still a viable threat.

The apparent reason for designing secure embedded software is multifold. First, companies want to protect their intellectual property to prevent competitors from catching up or gaining a competitive edge. Next, companies need to be protecting their users' data, which in itself can be worth a fortune. Finally, a high-profile attack that reaches the media can have catastrophic consequences to a brand that tanks the company stock or causes its users to switch to a competitor. (There are certainly others, but these are probably the biggest.)

Companies are deploying billions of IoT devices yearly. Unfortunately, these connected systems are often ripe for hackers interested in exploiting a device's data for financial gain, pleasure, or to prove their technical skills. There's hardly a day when there is no headline about the latest device hack, malware attack, or successful ransomware.

It's easy to think that your system will not be hacked. There are so many devices on the Internet; what are the chances that someone will find your device and target it? Well, the chances are pretty good! Several years ago, I worked on a project using a custom application processor running a Linux kernel. I was going to be traveling over the weekend and needed

remote access to the device to finish my work. The client agreed to provide remote access to the device. It went live on the Internet Friday night; by Sunday morning, the system had become part of a Chinese botnet!

Security is not optional anymore! Developers and companies must protect their intellectual property, customers' data, and privacy and meet the plethora of government regulations steadily being approved (recently, GDPR, NISTIR 8259A, the California SB-327, and the IoT Cybersecurity Improvement Act of 2019). Security must be designed into an embedded system from the very beginning, not added in at the end. Last-minute attempts to secure a device will only leave security vulnerabilities in the system that can be exploited.

In this chapter, we will start to explore how to design secure embedded applications. We'll start by exploring Arm's Platform Security Architecture. By the end of the chapter, you'll understand the fundamentals of how to perform a threat model and security analysis and the options you have to start designing your embedded software with security from the beginning.

## Platform Security Architecture (PSA)

Arm and ecosystem partners developed the **Platform Security Architecture** (PSA) to "provide the embedded systems industry with a baseline security architecture that could be leveraged to secure an embedded product."[1] PSA is designed to give developers holistic resources to simplify security. PSA consists of

- A set of threat models and security analysis documentation
- Hardware and firmware architecture specifications
- An open source firmware reference implementation[2]
- An independent security evaluation scheme

One way to look at PSA is as an industry best practice guide that allows security to be implemented consistently for hardware and software. Take a moment to absorb that statement; security is not just something done in software but also something in hardware! A secure solution requires both components to work together.

**Beware** Security requires an interplay between hardware and the software to successfully meet the systems security requirements.

It's important to realize that security is not just something you bolt onto your product when you are ready to ship it out the door. Instead, to get started writing secure software, you need to adopt industry best practices, start thinking about security from the beginning of the project, and ensure that the hardware can support the software needs. On multiple occasions, I've had companies request that I help them secure their products weeks before they officially launched. In several instances, their hardware did not support the necessary features to develop a robust security solution! Their security solution became a "hope and pray" solution.

Developing a secure solution using PSA includes four stages that can help guide a development team to constructing secure firmware. PSA guides teams to identify their security requirements up front so that the right hardware and software solutions can be put in place to minimize the device's attack surface and maximize the device's security. The four PSA stages include

- Analyze
- Architect
- Implement
- Certify

Figure 3-1 shows a complete overview of the PSA stages.

In stage 1, developers analyze their system requirements, identify data assets, and model the threats against their system and assets. In stage 2, Architect, developers select their microcontroller and associated security hardware in addition to architecting their software. In stage 3, Implementation, developers finally get to write their software. Developers can leverage PSA Certified components, processes, and software throughout the process to accelerate and improve software security. PSA Certified is a security certification scheme for Internet of Things (IoT) hardware, software, and devices.[3]

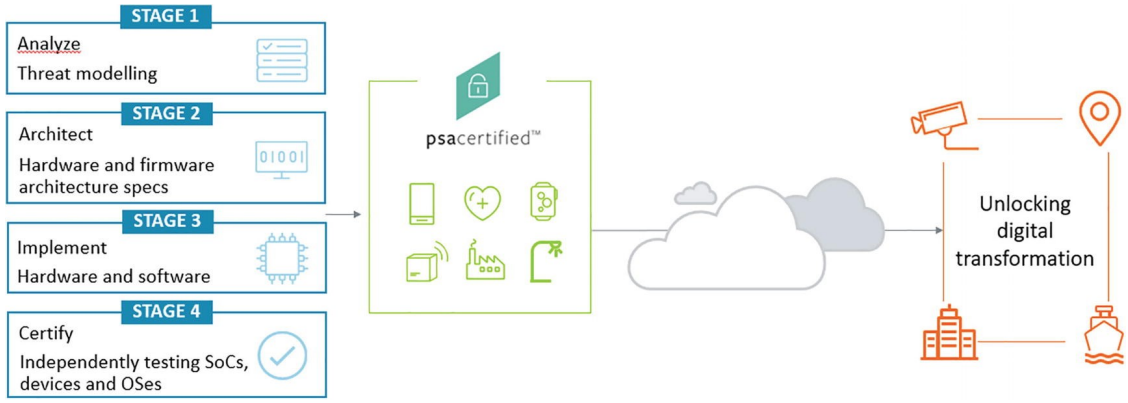


***Figure 3-1*** The four primary PSA stages and PSA Certified components help unlock a secure digital product transformation[4]

Since PSA's processes and best practices are essential to designing secure embedded applications, let's take a little more time to examine each PSA stage in a little more detail. Then, we will focus on the areas where I believe embedded developers will benefit the most, although each team may find specific areas more valuable than others.

## PSA Stage 1 – Analyzing a System for Threats and Vulnerabilities

When teams recognize that they need to implement security, they often find implementing security to be a nebulous, undefined activity. Teams know they need to secure the system but don't understand what they are trying to secure. Teams jump straight into concepts like encryption and secure communications in many cases. Encryption on its own is not going to secure a system, it might secure communications, but the system can still be vulnerable to wide variety of attacks.

Designing a secure embedded system starts with identifying the assets that need to be protected and the threats those assets will face. Once the assets and threats are identified, developers can properly define their security requirements to protect them. The security requirements will then dictate the security strategies and tactics used and the hardware and software needed to protect those assets adequately. Formally, the process is known as a threat model and security analysis (TMSA).

A TMSA that follows the best practices defined in PSA has five distinct steps that can be seen in Figure 3-2. These include defining assets, identifying

adversaries and threats, defining security objectives, writing security requirements, and summarizing the TMSA results.


***Figure 3-2*** The five steps developers walk through to perform a TMSA. Image Source: (Arm)[5]

Thoroughly discussing how to perform a TMSA is beyond the scope of this book,[6] but let's examine a few high-level concepts of what a TMSA might look like for an Internet-connected weather station.

### TMSA Step #1 – Identifying Assets

The first step in the TMSA is to identify assets that might need to be protected. It's interesting to note that most assets that need to be protected in a system are data assets under software control. For example, an IoT device will commonly have data assets such as

- A unique device identification number
- The firmware[7]
- Encryption keys
- Device keys
- Etc.

Every product will also have unique data assets that need to be protected. For example, a biomedical device might protect data such as

- A user profile
- Raw sensor data
- Processed sensor data
- System configuration
- Control data
- Etc.

A weather station might protect data such as

- Location
- Local communication interface(s)

- Raw sensor data
- System configuration
- Remote access credentials
- Etc.

This is an excellent time to point out the parallels between identifying data assets from a security perspective with the design philosophies we outlined in Chapter 1. We have already established that data dictates design. In many instances, our data is a data asset that we would identify in this TMSA step. We have outlined the design philosophies that encompass ideas and concepts for secure application development, including that security is king.

Another essential point to consider is that every identified asset in our system doesn't have to be protected. There will be some assets that will be public data that require no security mechanisms. Trying to protect those publicly available assets can decrease the system's security! Imagine taking publicly known data and then encrypting it in a data stream. We can use this information to break the encryption scheme if we know the public data is in the stream and what the data is. Only protect the assets that need to be protected.

**Best Practice**Only protect data assets that need to be protected. "Overdoing it" can lead to vulnerabilities just as much as not doing anything. Carefully balance based on your system needs.

### TMSA Step #2 – Identifying Adversaries and Threats

Once the data assets that need to be protected have been identified, the developers can specify the adversaries and the threats to the assets. Identifying adversaries can be as simple as creating a list. Still, I often like to include characteristics of the adversary and rank them based on the likelihood and the impact the adversary may have on the product and the business. Table 3-1 shows an example of what an adversary analysis might look like.

***Table 3-1*** Example adversary analysis that includes a risk assessment from each

| Adversary | Impact | Likelihood | Comments |
|---|---|---|---|

| Adversary | Impact | Likelihood | Comments |
|---|---|---|---|
| Competitor | High | High | IP theft |
| End user | Low | Medium | Feature availability |
| Hackers | Medium | Medium | Data theft, malware |
| Nation state | High | Low | Data theft, malware |
| Internal personnel | High | Medium | IP and data theft |

Each adversary will have its purpose for attacking the system. For example, an adversary may be interested in the data on the system, overriding controls, spying, stealing intellectual property, or any number of other reasons. Just because an adversary exists does not necessarily mean that someone will be interested in your system (although assuming no one is interested is super dangerous). Developers should rank each adversary based on the risk (impact) to the business, user, and system and determine the likelihood of an attack. Figure 3-3 shows a matrix often used to evaluate risk.

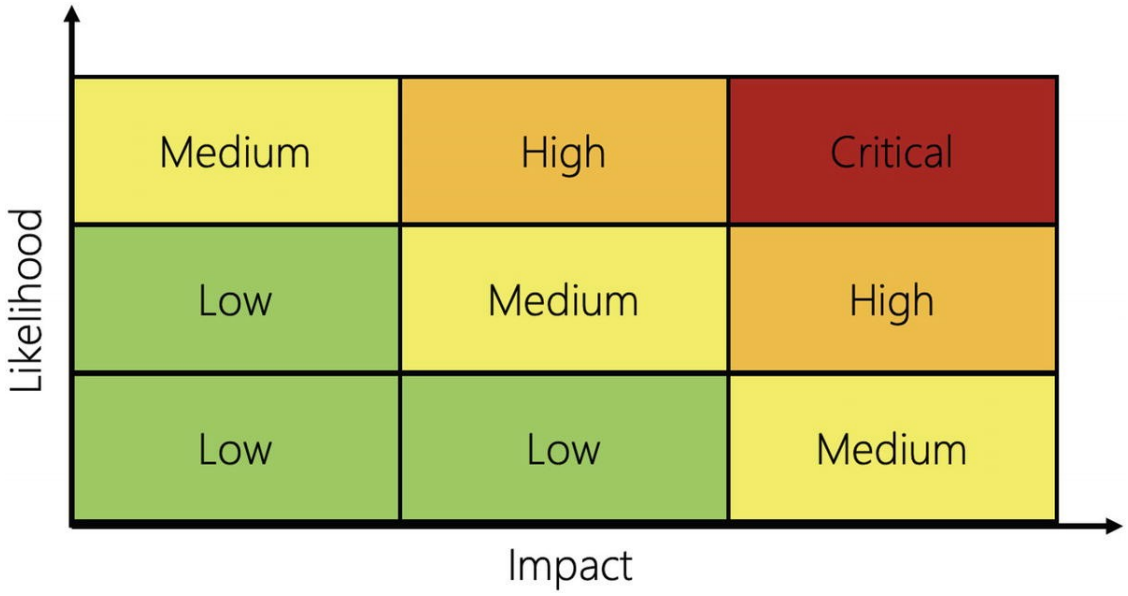


***Figure 3-3*** A standard risk matrix trades off impact and the likelihood of an event[8]

From a quick glance, Figure 3-3 gives us the foundation for developing our own risk matrix. Unfortunately, the generalized matrix doesn't really tell you much. Instead, I would recommend that you take the format and then update it for your

specific situation. For example, if I take Table 3-1 and overlay it on the risk matrix, the result will be something like Figure 3-4. Figure 3-4 helps to create a better picture of what we are facing and hopefully allows us to generate some actionable behaviors from it.

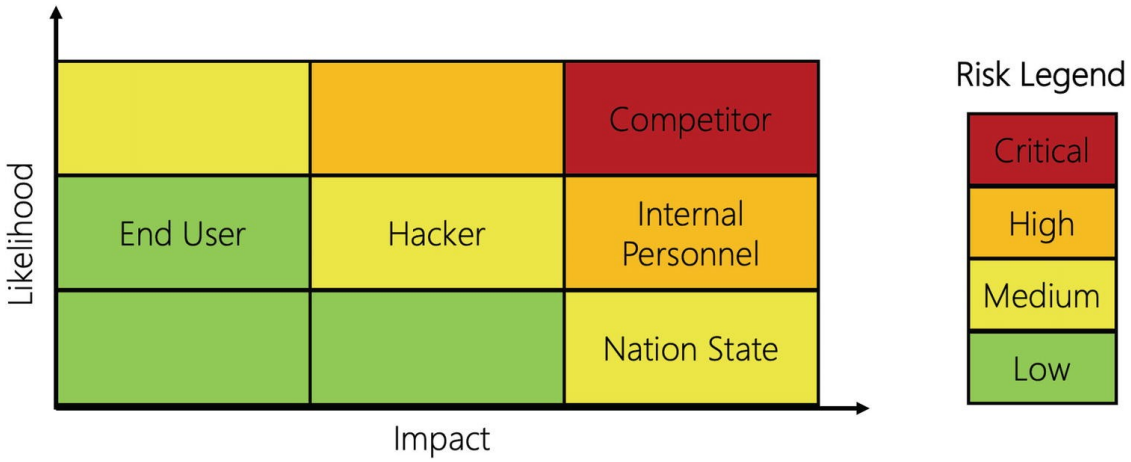


***Figure 3-4*** An actionable risk table that demonstrates the risk for potential adversaries

We can see that for our hypothetical system, the highest risk is from our competitors followed by our internal personnel. Now that we understand who might be attacking our system, we can decide where to focus our security efforts. If the end user will have minimal impact and likelihood, we may just put enough security in place to make it not worth their time. Instead, we focus on making sure our competitors can't steal our intellectual property or gain a competitive advantage.

Adversaries will target and attack a system's assets in various ways; these attacks are often referred to as threats. Every embedded system will face different and unique threats, but there will be a standard set of threats. For example, an adversary interested in gaining access to the system might attack the system's credentials through an impersonation attack. Another adversary may attack the firmware to inject code or modify its functions.

TMSA recommends developers identify the various threats that each asset will face. Teams often find it helpful to build a table that lists each asset and includes columns for various analytical components. For example, Table 3-2 shows several assets and the expected threats for a connected weather station.

***Table 3-2*** A list of assets and their threats for a fictitious connected weather station[9]

| Data Asset | Threats |
|---|---|
| Location | Man-in-the-middle, repudiation, impersonation, disclosure |
| Raw sensor data | Tamper (disclosure) |
| System configuration | Tamper (disclosure) |
| Credentials | Impersonation, tamper (disclosure), man-in-the-middle, escalation of privilege (firmware abuse) |
| Firmware | Tamper, escalation of privilege, denial of service, malware |

As the reader may imagine, it can also be advantageous to reverse this table and list threats that list the data assets. I won't display such a table for brevity, but you can see an expanded version in Table 3-3. I'll only recommend that both tables be created and maintained.

**TMSA Step #3 – Defining Security Objectives**

The TMSA helps developers and teams define the security objects for their systems. There are typically two parts to the security objective definitions. First is identifying which combination of confidentiality, integrity, and authenticity will protect the assets. Second is the security objective that will be used to secure the asset. Let's examine each of these two areas separately.

Confidentiality, integrity, and authenticity are often called the CIA model.[10] Every data asset that a team is interested in protecting will require some combination of CIA to protect the asset. For example, credentials would require confidentiality. The system configuration might require confidentiality and integrity. The product's firmware would require

confidentiality, integrity, and authorization. Before we can define the combination that fits an asset, it's helpful to understand what each element of the CIA does.

Confidentiality indicates that an asset needs to be kept private or secret. For example, data assets such as passwords, user data, and so forth would be confidential. Access to the data asset would be limited to only authorized users. Confidentiality often indicates that encryption needs to be used. However, that encryption often goes beyond just point-to-point encryption that we find in the connection between the device and the cloud. Confidential assets often remain encrypted on the device.

Integrity indicates that the data asset needs to remain whole or unchanged. For example, a team would want to verify the integrity of their firmware during boot to ensure that it has not been changed. The transfer of a firmware update would also need to use integrity. We don't want the firmware to be changed in-flight. Data integrity is often used to protect the system or a data asset against malware attacks.

Authenticity indicates that we can verify where and from whom the data asset came. For example, when performing an over-the-air (OTA) update, we want the system to authenticate the firmware update. Authentication involves using a digital signature to verify the integrity of the asset and verify the origin of the asset. Authenticity is all about confirming the trustworthiness of the data asset.

When performing a TMSA, it is relatively common to analyze each data asset and assign which elements of the CIA model will be used to protect it. Of course, the exact methods will often vary, but Table 3-3 shows a simple way to do this using the assets we have already defined using a weather station as an example.

***Table 3-3*** A list of assets, threats, and the CIA elements used to protect the asset. C = Confidentiality, I = Integrity, and A = Authenticity

| Data Asset | CIA | Threat |
|---|---|---|
| Location | C, I | Man-in-the-middle, Repudiation, Impersonation, Disclosure |
| Raw Sensor Data | C, I | Tamper (Disclosure) |
| System Configuration | C, I | Tamper (Disclosure) |
| Credentials | C, I | Impersonation, Tamper (Disclosure), Man-in-the-middle, Escalation of Privilege (firmware abuse) |
| Firmware | C, I, A | Tamper, Escalation of Privilege, Denial of Service, Malware |

Once we understand the data assets and the CIA model elements we need to use to protect them, we can define the security objectives for the data asset. Of course, we may want to implement many security objectives. Still, the most common in an embedded system are access control, secure storage, firmware authenticity, communication, and secure storage. The following is a list of definitions for each:[11]

**Access control** – The device authenticates all actors (human or machine) attempting to access data assets. Access control prevents unauthorized access to data assets. It counters spoofing and malware threats where the attacker modifies firmware or installs an outdated flawed version.

**Secure storage** – The device maintains confidentiality (as required) and integrity of data assets. This counters tamper threats.

**Firmware authenticity** – The device verifies firmware authenticity before boot and upgrade. This counters malware threats.

**Communication** – The device authenticates remote servers, provides confidentiality (as required), and maintains the integrity of exchanged data. This counters man-in-the-middle (MitM) threats.

**Secure state** – This ensures that the device maintains a secure state even in case of failure to verify firmware integrity and authenticity. This counters malware and tamper threats.

Usually, once the security objectives are outlined, the team will discover that a single security objective will often cover multiple data assets. Therefore, it can be helpful to list the security objectives, threats, and data assets covered by the objective.

With the objectives outlined, the next step in the TMSA is to define the security requirements.

**TMSA Step #4 – Security Requirements**

Once a team understands their data assets, threats, and the objectives to protect those assets, they can then define their security requirements. Security requirements are more interesting than standard software requirements because security is not static throughout the product life cycle. The system may have different security requirements at various stages of development, such as during design, manufacturing, inventory, normal operation, and end of life.

As part of the security requirements, teams should outline how their requirements may change with time. You might think I just want to set my requirements and never change them. However, this problem can add a lot of extra heartache at specific points during development. For example, a team will not want to disable the SWD/JTAG interface during design and implementation. If they do, debugging will become a slow, hellish nightmare. Instead, they want to ensure the SWD/JTAG port is disabled when the devices enter the field.

A simple example to consider is the configuration data asset. We want to protect the asset from spoofing using access control and confidentiality security property. We don't have a security objective or requirement during design, manufacturing, and while the product sits in the inventory. However, once the product is in a user's hands, we want the system configuration to be encrypted. At the end of life, we want the configuration scrubbed and the microcontroller placed into a non-operational state.

I highly recommend taking the time to think through each asset at each stage of development to properly define the security requirements. These requirements will be used to identify hardware and software assets necessary to fulfill and secure the system.

**TMSA Step #5 – Summary**

The last step in the TMSA process is to summarize the results. A good TMSA will be summarized using several documents, such as a detailed TMSA analysis document and a TMSA summary.

The detailed TMSA analysis document will walk through the following:

- Explain the target of evaluation
- Define the security problem
- Identify the threats
- Define organizational security policies
- Define security objectives
- Define security requirements

The TMSA overview is often just a spreadsheet that summarizes the assets, threats, and how the assets will be protected. An example summary from the Arm TMSA for the firmware data asset for an asset tracker can be found in Figure 3-5.

This spreadsheet is an "at a glance" summary of a Threat Model and Security Analysis (TMSA) document (DEN0075). It is a quick-reference summary of assets, threats, impact and security requirements that are provided in Excel format to allow you to easily edit and extend. We hope that you find this document useful as a starting point.

arm



CI=Arm CryptoIsland, PSA=Arm Platform Security Architecture

| Asset | Security Property | Threat | Entry Point of Threat (where the attack is launched from ex: malware, network, JTAG, etc) | Impact of Vulnerability | Severity (CVSS Rating) | Mitigation/Security Requirement | Arm's Technology |
|---|---|---|---|---|---|---|---|
| Firmware | Integrity | Tamper | Malware, bug, mass storage access, JTAG, network, update | Install malware | Critical: 9 CVSS:3.0/AV:N/AC:H/PR:N/UI:N/S:C/C:H/I:H/A:H | Support secure boot flows (authenticate firmware) | [CI] Loaded SW validation functionality |
| | | Escalation of privilege (Firmware Abuse) | | Launch DDoS | | Enforce principle of least privilege | |
| | | | | | | Support secure firmware update (authenticate update) | [CI] SW update validation |
| | | | | | | | [PSA] Trusted Boot features |
| | | DoS (Firmware Abuse) | | Tamper/Steal location | | Support anti-rollback of firmware | [CI] SW update validation |
| | | | | Permanent bricking of device | | | [PSA] Firmware Update features |

***Figure 3-5*** A TMSA summary for a firmware data asset (Source: psacertified.org)

There are several TMSA examples that Arm has put together that can be found **here**.[12] The PSA website also has many resources for getting started with a TMSA.[13] If you plan to perform your TMSA, I would highly recommend that a professional perform your TMSA or have a security professional review your results. Several times, I've gone to review a client's TMSA and found some wide gaps, holes, and oversights in their analysis.

## PSA Stage 2 – Architect

Once we understand our data assets, threats, security properties, objectives, and requirements, we are ready to move to the second PSA stage, architecting our software. The key to securing an embedded system is through isolation. Isolation is the foundation on which security is built into an application. It separates software modules, processes, and memory into separate regions with minimal or no interaction with each other. The isolation acts as a barrier limiting the attack surface that an attacker can leverage to access an asset of interest on the device.

### Security Through Isolation

There is an easy way to think about the benefits of isolation in a secure application. Consider an office building where the office is organized as an open floor plan. Everyone's desks are out in the open, and everyone can get access to everyone else's workspace. The building has two security access points: a front and a back door. An attacker needs to circumvent the front or back door security to gain access to the entire building. They have access to the whole building to accomplish this, as shown in Figure 3-6.

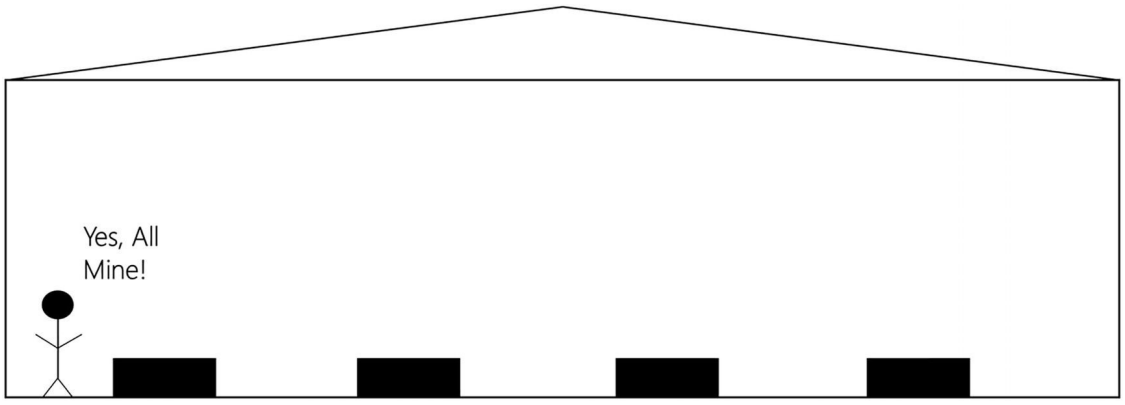


***Figure 3-6*** An adversary can bypass security at the front and access the entire building

Now, consider an office building where the office is organized into private offices for every employee. Each office has an automatic lock that requires a keyed entry. On top of this, building security is still at the front and back doors. If an attacker can breach the outer security, they have access to the building but don't have access to everyone's workspace! The doors isolate each office by providing a barrier to entry, as shown in Figure 3-7. The attacker would now need to work harder to access one of the offices, which only gives them access to one office, not all of them.

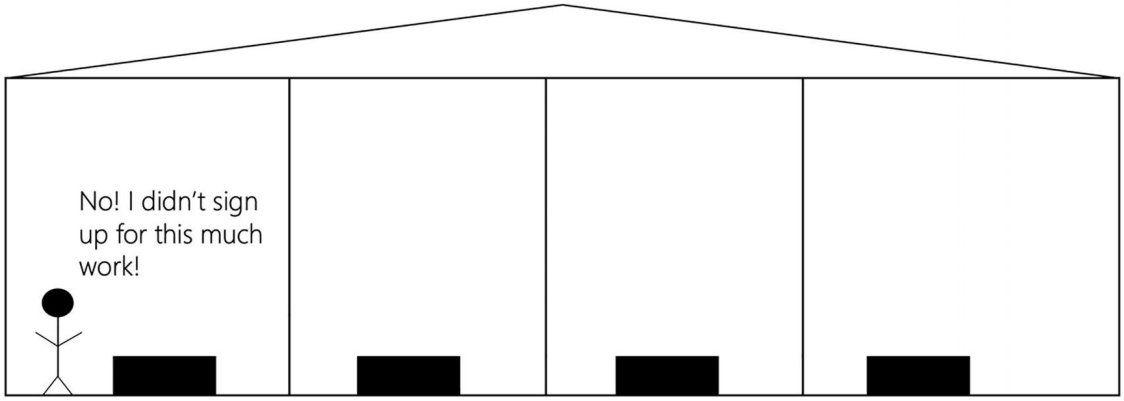


***Figure 3-7*** An adversary can bypass security at the front or back, but they don't have free reign and have additional work to access critical assets

The key to designing secure embedded software is to build isolation into the design that erects barriers to resources and data assets. Then, if hackers find their way into the system, they are provided minimal access to the system with essential assets, hopefully further isolated.

Take a moment to consider how designers currently architect most microcontroller-based embedded systems. For example, it's not uncommon to find applications running in privileged mode with the entire application written into one giant memory space where every function can access all memory and resources without constraint. As you can imagine, this results in a not very secure system.

A secure application design must break the application into separate tasks, processes, and memory locations where each has limited access to the data assets and system resources. In addition, a successful, secure application design requires that the underlying hardware support hardware-based isolation. There are three levels of isolation that designers need to consider: processing environments, Root-of-Trust with trusted services, and trusted applications. An overview of these three levels can be seen in Figure 3-8.

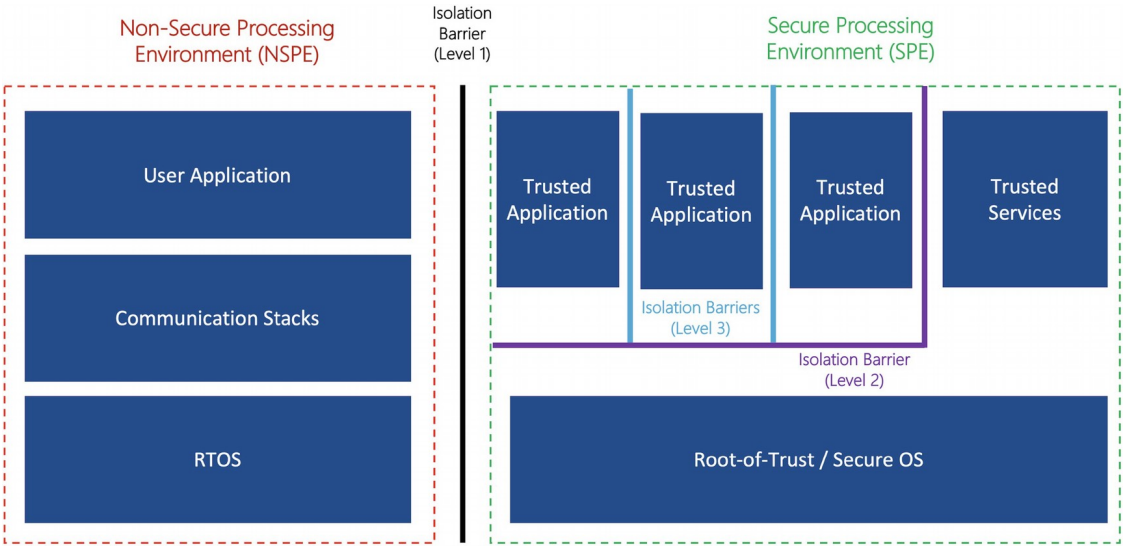


***Figure 3-8*** Securing an embedded application requires three levels of isolation: isolated processing environments, an isolated Root-of-Trust with trusted services, and application isolation

### Isolation Level #1 – Processing Environments

The first layer of isolation in a secure application separates the execution environments into two isolated hardware environments: the secure processing environment (SPE) and the nonsecure processing environment (NSPE). The SPE, also known as the trusted execution environment (TEE), contains the Root-of-Trust (RoT), trusted services like cryptographic services, and trusted application components. The NSPE, also known as the rich execution environment, includes the typical software developers' use, like an RTOS, and most of their general application code.

A designer can use two mechanisms to create the first level of isolation: multicore processors and Arm TrustZone. The designer sets one of the processing cores in the multicore solution to act as the NSPE and the second core to serve as the SPE. The two cores would be isolated with some shared memory and interprocessor communications (IPC). The secure core would only allow a minimal set of operations publicly available to the NSPE. One advantage to using multiple processors is that the NSPE and the SPE can execute code simultaneously.

Arm TrustZone is a solution that provides hardware-based isolation between the NSPE and the SPE, but it is done in a single-core solution. When an SPE function is called, the processor switches from the NSPE environment to the SPE environment in deterministic three clock cycles. When

the SPE is complete, the processor switches back to the NSPE. We will explore a little bit more about these environments later.

### Isolation Level #2 – Root-of-Trust and Trusted Services

The second level of isolation in a secure application is established in the SPE through a Root-of-Trust (RoT) and trusted services.

A Root-of-Trust provides the trust anchor in a system to support the secure boot process and services.[14] A Root-of-Trust should be immutable and hardware based to make it immune to attack. The Root-of-Trust often includes hardware-accelerated cryptography, true random number generation (TRNG), and secure storage.

A good Root-of-Trust will start with the chip manufacturer. For example, the Cypress PSoC 64, now part of Infineon, ships with a hardware-based Root-of-Trust from the factory, with Cypress owning the Root-of-Trust. As part of the Root-of-Trust, Cypress has a unique identifier for each chip with Cypress keys that allow a user to verify the authenticity and integrity of the hardware. Developers can then verify the chip came from Cypress and transfer ownership of the Root-of-Trust to their company. From there, they can inject their user assets, provision the device, and so forth.

Trusted services also reside in the SPE. There are many types of services that can be supported. For example, it is not uncommon for trusted services to include a secure boot process through a chain of trust, device provisioning, attestation, secure storage, Transport Layer Security (TLS), and even firmware over-the-air (FOTA) updates. These are all standard services that nearly every connected needs to build a secure application successfully.

### Isolation Level #3 – Trusted Applications

Trusted applications are the developer's applications that run in the SPE in their own isolated memory space. Trusted applications provide services to the system that require secure access to data and hardware re-

sources.

A trusted application is built upon the Root-of-Trust and other trusted services. In addition, trusted applications are often implemented by leveraging the capabilities provided by an open source framework called Trusted Firmware M (TF-M). TF-M is a reference software platform for Arm Cortex-M processors that implements SPE and provides baseline services that trusted applications are built upon, such as

- Secure boot
- Managing isolation
- Secure storage and attestation services

One of the goals of TF-M is to provide a reference code base that adheres to the best practices for secure systems defined by the Platform Security Architecture (PSA).

### Architecting a Secure Application

Architecting a secure application requires the developer to create layers of isolation. The process starts with separating the processing environments, isolating the RoT and trusted services, and then the trusted applications. What I often find interesting is that developers architecting a solution try to jump straight into microcontroller or processor selection! They act as if the microcontroller dictates the architecture when it's the other way around! So at the architectural stage, we architect the best security solution for our product and let the implementers figure out the details.[15]

I highly recommend that embedded software architects create a model of their secure application. The model doesn't have to be a functional model per se, but something that is thought through carefully and diagrammed. The model should

- Identify what goes in each processing environment
- Break up the system into trusted applications
- Define the RoT and trusted services
- Identify general mechanisms for protection such as MPUs, SMPUs, etc.

- Define general processes and memory organization (in a generic way)

A simplified example of what this might look like can be found in Figure 3-9. Usually, this would also break out references to the TMSA and other services, components, and more. Unfortunately, such a diagram would be unreadable in book format. However, the reader gets a general idea and should extrapolate for their application design.

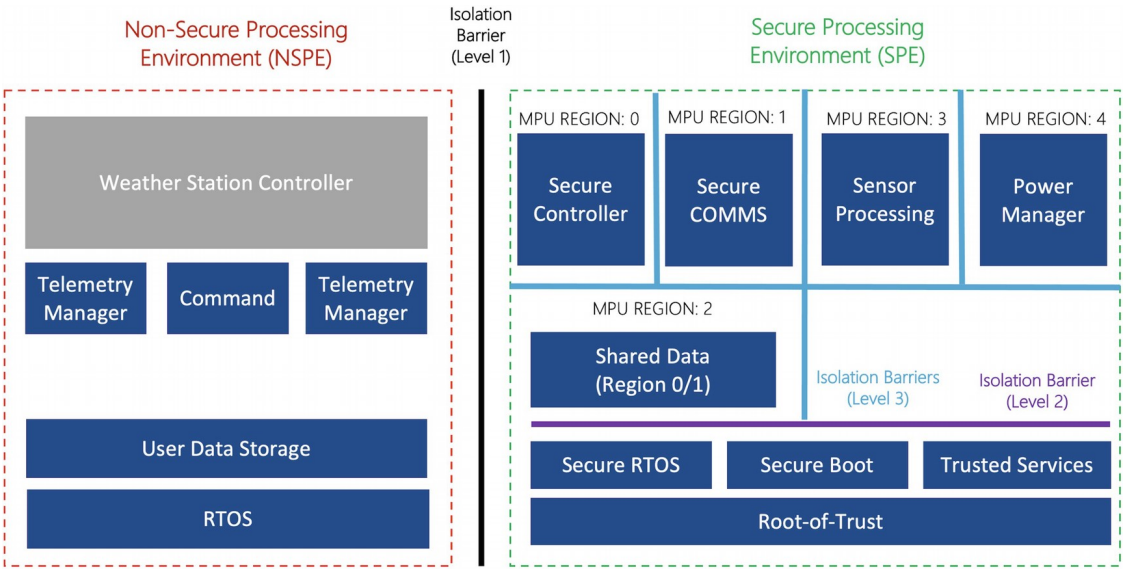


***Figure 3-9*** A secure application model lays the foundation for the application developers to understand the security architecture and designers' intentions

When you architect your security solution, it's essential to realize that you aren't just looking at where different components and pieces of code and data should be. The end solution may be using a real-time operating system (RTOS) which will also require developers to carefully think through where the various application threads should be located and which state the processor will be in to execute the functions within those threads. There are three different types of threads that architects and developers should consider:

- **Nonsecure threads** only call functions that exist within the NSPE.
- **Secure threads** which only call functions that exist within the SPE.
- **Hybrid threads** execute functions within both the SPE and the NSPE. Hybrid threads often need access to an asset the architect is trying to protect.

A simple UML sequence diagram can be architected how the application will behave from a thread execution standpoint. The sequence diagram

can show the transition points from the NSPE to the SPE and vice versa. The sequence diagram can dramatically clarify a developer's understanding of the implementation and how hybrid threads will transition between the NSPE and the SPE. Figure 3-10 demonstrates what an example hybrid thread sequence diagram might look like.

The diagram contains two threads, a sensor, and a communications (comm) thread. Certain operations for each thread are executed within the NSPE and the SPE. We can see the general operations blocks at specific points along the thread's lifeline. Again, this is just another example of what an architect might do to help clarify the intent of the design so that developers can implement the solution properly.

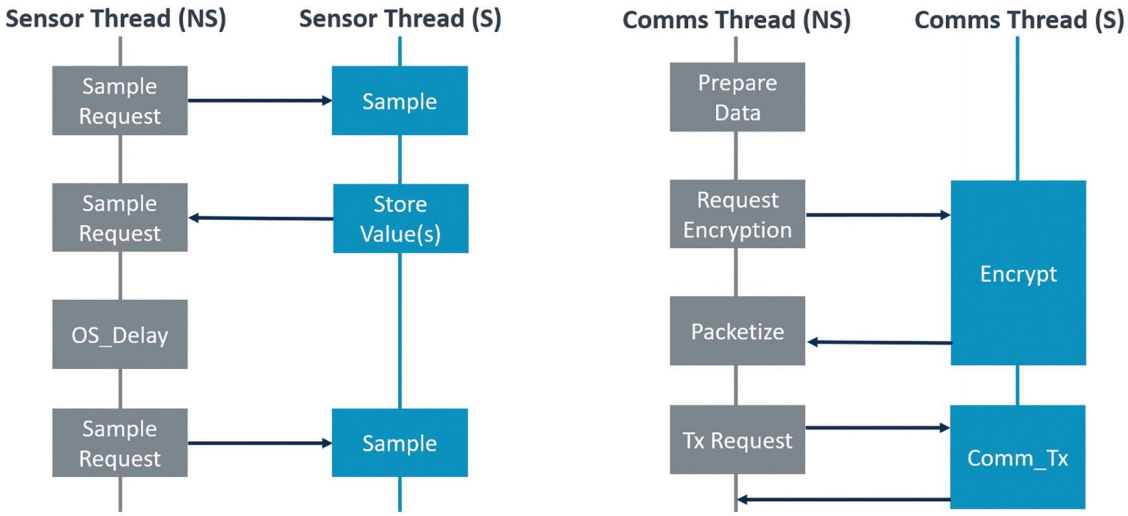


***Figure 3-10*** A sequence diagram can demonstrate transition points in a thread-based design between NSPE and SPE execution[16]

## PSA Stage 3 – Implementation

The third stage of PSA is all about implementation. Implementation is where most developers' minds go first and are most tempted to begin. However, as stated before, we need to have our architecture in place first to implement a secure application successfully.[17] In general, two methods are used in microcontroller-based systems to implement and isolate the runtime environments that don't require an external security processor: multicore processors and Arm TrustZone.

### Multicore Processors

Multicore processors provide developers unique and interesting solutions to create an SPE and NSPE. Multicore solutions are typically dual-core solutions with one core dedicated to the nonsecure processing environment (NSPE). The second is dedicated as a security processor that runs the secure processing environment (SPE). My prediction is that over the next decade, or so, we will start to see microcontrollers with more than two cores. Multiple cores will provide multiple hardware-based execution partitions that would increase the security capabilities of processors.

Today, a typical multicore architecture has two processing cores that may or may not be the same processor architecture. For example, the Cypress PSoC 64 series have an Arm Cortex-M0+ processor that runs the SPE and an Arm Cortex-M4 processor that runs the NSPE. Each core will also access its isolated memory regions and an interprocessor communication (IPC) bus. The IPC is used to communicate between the two. Generally, the NSPE will request an operation from the SPE, which the SPE will perform. The general architecture for a multicore solution can be seen in Figure 3-11.

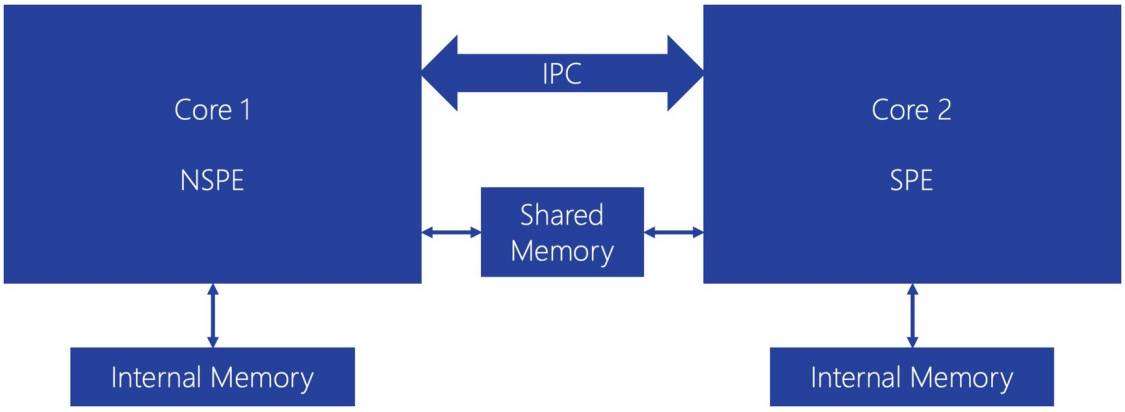


***Figure 3-11*** A multicore microcontroller can set up an SPE in one core while running the NSPE in the other. Each core has dedicated isolated memory, shared memory for data transfer, and interprocessor communications (IPC) to execute secure functions

The multicore solution does have several advantages and disadvantages, just like any engineering solution. One advantage is that each core can execute application instructions in parallel, improving the application's responsiveness and performance. For example, if data needs to be encrypted, the cryptographic operation can be done in the SPE while the NSPE is still processing a touch screen and updating the display. However, it's not uncommon for developers to struggle with performance issues when implementing their security solutions, especially if they did not se-

lect their microcontroller hardware properly (see Chapter 11).

There are several other advantages to using a multicore processor as well. For example, having both cores on a single die can decrease costs and limit the attack surface provided by an external security processor. Each core is also isolated, which limits access and adds additional complexity to trying to exploit the system.

Multicore solutions also have some disadvantages. First, the cost of a multicore microcontroller tends to be higher than single-core solutions. However, this may not always be the case, as economics drives these prices, such as volumes, demand, and so forth. Writing multicore applications can also be more complex and costly and require more debug time. However, developers should minimize what they put into the SPE to reduce their attack surface, limiting how much extra complexity is added.

### Single-Core Processors with TrustZone

Starting with the Armv8-M architecture, silicon providers have the opportunity to include hardware-based isolation within a single-core processor through the use of TrustZone technology. TrustZone offers an efficient, system-wide approach to security with hardware-enforced isolation built into the CPU.[18] I mention "have the opportunity to" because not all Armv8-M processors and beyond have TrustZone. Furthermore, it's optional, so if you want the technology, you must be careful about the microcontroller you select to ensure it has it.

The idea behind TrustZone is that it allows the developer to break their processors' memory up into an NSPE and SPE. Flash, RAM, and peripheral memory can all be assigned to either NSPE or SPE. In addition, developers can leverage CMSIS-Zone,[19] which creates a configuration file that allows developers to partition their system resources across multiple projects and processors. (Yes, for secure applications, you will have to manage one project for the NSPE and one for the SPE!) CMSIS-Zone includes an Eclipse-based utility that provides a simple GUI to[20]

- Assign resources
- Display all available system resources, including memory and pe-

ripherals

- Allow the developer to partition memory and assign resources to subsystems
- Support the setup of secure, nonsecure, and MPU-protected execution zones with the assignment of memory, peripherals, and interrupts
- Provide a data model to generate configuration files for tool and hardware setup

Developers should keep in mind that TrustZone is a single-core solution. Unlike the multicore solution, it can only execute the SPE or the NSPE simultaneously. For example, if the NSPE is running and an operation is requested that runs in the SPE, deterministic three clock cycles occur to transition the hardware from NSPE to SPE. Once in the SPE, another setup may occur based on the application before the SPE runs the operation. When the procedure is complete, the processor scrubs CPU registers (not all of them, so see your documentation) and then transitions back to the NSPE.

The boot sequence for a TrustZone application is also slightly different from multicore. In multicore, the developer can bring both cores up simultaneously or bring the security core and validate the NSPE first. Figure 3-12 shows the typical TrustZone application boot and execution process. The boot process starts in the SPE, allowing the secure boot and bootloader to execute and validate the NSPE before jumping to the NSPE for execution.

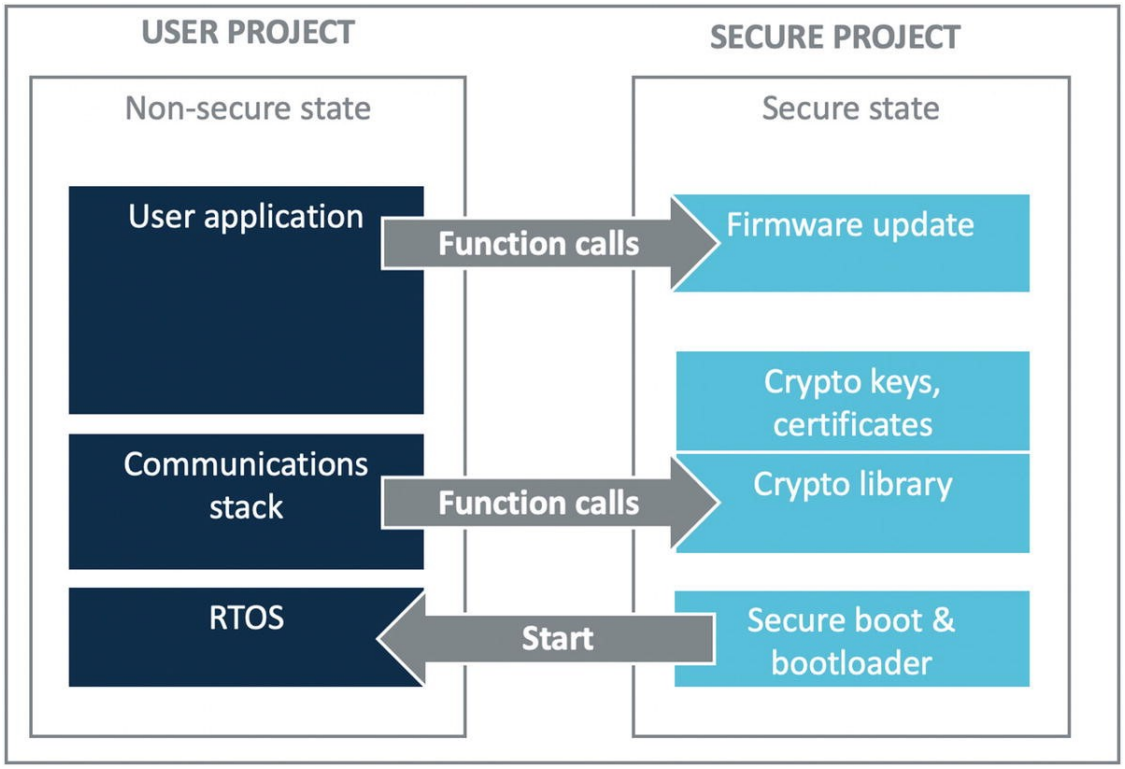


***Figure 3-12*** A TrustZone application boots into the SPE, validates the NSPE, and executes the user project

While the user project executes its application, it will make function calls into the SPE through a secure gateway. A secure gateway is an opening in the SPE that only allows access to function calls. For example, the secure state project can expose the function calls for the cryptographic library. The user project can then make calls to those functions, but only those functions! An exception will be thrown if a developer tries to create a pointer and access memory that is not exposed by a secure gateway. The only area of the SPE that the NSPE can access is the functions exposed by the SPE. However, the SPE can access all memory and functions, including the NSPE! Therefore, developers must be cautious with what they access and where those accesses are.

Single-core processors do have the advantage of being less expensive than multicore processors. However, they can also access the entire memory map, which is a disadvantage. One drawback is that clock cycles are wasted when transitioning from NSPE to SPE and SPE to NSPE. Admittedly, it's not very many clock cycles, but those few clocks could add up over time if the application is concerned with performance or energy consumption.

### The Secure Boot Process

The boot process I see companies using in the microcontroller space generally leaves much, much desired. The typical boot process I encounter verifies an application checksum, and that's about it. However, a secure boot process needs must more than just confirming that the application's integrity is intact. A secure boot process involves establishing a Root-of-Trust by verifying the onboard chip certificate and then booting the application in stages. Each boot stage verifies the authenticity and integrity of the firmware images that will be executed.

Figure 3-13 shows what is typically involved in the boot process. As you can see, this is much more involved than simply jumping to a bootloader, verifying the application is intact through a checksum or CRC, and then running the application. At each step, before loading the next application in the boot process, the new image is validated. For our purposes, validation is verifying the integrity of the image and verifying the authenticity of the image. The application must come from the right source and also be intact. Developers implementing a secure application need to ensure they develop a Chain-of-Trust throughout the entire boot process where the integrity and authenticity of each loaded image are verified.

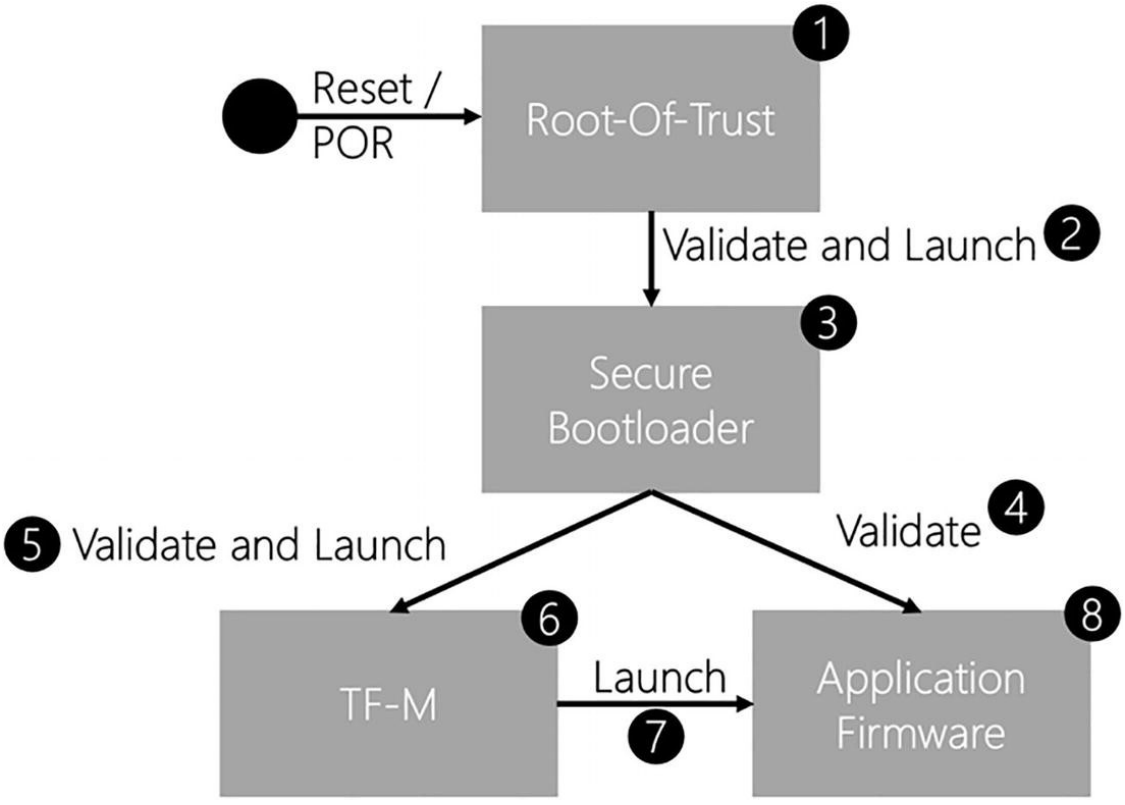


***Figure 3-13*** A Chain-of-Trust is often used to securely boot a system where each stage is verified through authenticity and validated for integrity before loading and executing the image

Developing a Chain-of-Trust involves several steps for an embedded application. First, the Root-of-Trust is a trusted anchor from which all other trusts in the system originate. I often recommend that developers find microcontrollers that have a hardware-based Root-of-Trust. In these cases, the microcontroller manufacturer often creates a Root-of-Trust and transfers it to the developing company.[21] The developing company can then add their credentials and build their system.

Second, the RoT during boot will authenticate and validate the next application image. In most cases, the following application is a secure bootloader. The RoT will use a hash to validate the secure bootloader. Once validating, the RoT will launch the bootloader. The bootloader will then check for any new firmware that is being requested to load. If so, the firmware update process will occur (which is beyond our scope now); otherwise, the existing application images will be used.

When not updating firmware, the secure bootloader has two jobs that it must complete: validate the existing application firmware (NSPE user application) and then validate and launch the secure firmware (SPE). The secure bootloader does not launch the user application that exists in the NSPE. Instead, the SPE, which contains the secure runtime, will launch the NSPE once it has been deemed that the system is secure and the NSPE is ready to launch.

The secure firmware in Arm Cortex-M processors is often based on Trust Firmware for Cortex-M (TF-M) processors. TF-M is an open source reference implementation of the PSA IoT Security Framework.[22] TF-M contains common resources across security application implementations, such as secure storage, attestation, and cryptographic operations. If the user application that runs in the NSPE is validated successfully, it will launch and allow the system to operate.

### PSA Stage 4 – Certify

How do you know that the components you are using in implementation and your system meet its security requirements? One potential answer is that you certify your device. Certification is the last stage in PSA, and it's

also the one stage that is potentially optional (although I always recommend that even if you don't plan to certify your device or software, you at least hire an expert to perform penetration testing on it).

Certification can be performed on both hardware and software. Usually, PSA certification is focused on component or hardware providers who want to certify their components for companies that will develop products. However, consumer-focused devices can also be certified. Certification guarantees that the developers considered and implemented industry-recommended best practices when they designed and implemented their system.

PSA Certified is an independent security evaluation scheme for IoT chips, operating systems, and devices. There are several lab companies that can perform the certification process.[23] Each lab is overseen by TrustCB, which is the company that certifies the labs and was involved in the creation of PSA. The time and costs required to certify will vary depending on the level of certification sought and what is being certified. For example, companies can certify for their hardware device, silicon chips, system software, and IP providers.

PSA Certified provides a much-needed scheme for microcontroller-based devices, plus a common language that the whole industry can use and understand. PSA Certified contains three different levels, which increase the security robustness at each level. Figure **3-14** shows the three different general ideas behind each.

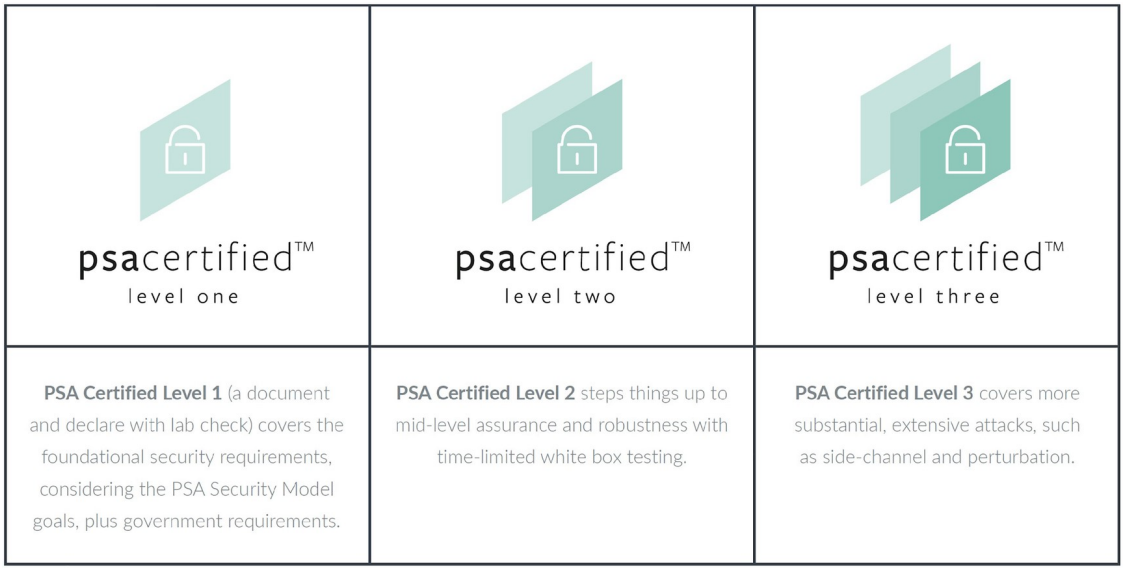

***Figure 3-14*** PSA Certified provides three certification levels that increase the robustness of hardware or software at each level

## Final Thoughts

We are entering an age for embedded software where developers can no longer ignore security. If a device connects to the Internet, then the system must take security into account. If the system is not connected, well, you still might want to add basic security like tamper protection to protect your intellectual property. Security can be a touchy subject because the threats are always evolving, and many embedded software developers don't have the knowledge or experience with designing and building secure software.

Developers can leverage existing frameworks like PSA to help guide them through the process. There is a big push right now to simplify security for embedded systems. Software frameworks like TF-M can help teams get a jump-start in deploying trusted services to their devices. If there is anything that this chapter has taught you, I hope that it is that you can't bolt security onto your system at the end. Security must be designed into the system from the very beginning, or else the system may be left vulnerable to attack.

Don't wait until your system and company make headlines news! Start thinking about security today and with every line of code you write!

**Action Items**
To put this chapter's concepts into action, here are a few activities the reader can perform to start applying secure application design principles to their application(s):

- Explore the many articles and white papers that explain PSA. A good starting point is
  - `https://developer.arm.com/architectures/architecture-security-features/platform-security`
- Perform a TMSA on an existing product or a fictitious product for practice. Leverage the PSA TMSA examples to get started.
- Investigate TF-M or a similar platform that is used in a security appli-

cation. First, review the APIs, functions, and capabilities. Then, how can you use these in your secure applications?

- Practice developing a secure architecture. Either use an existing product or a fictitious product to practice decomposing an application into NSPE and SPE.
  - How can the application be broken up into components?
  - Where should the application components live?
- Download and install CMSIS-Zone. Explore and build an example system to understand how to partition applications and memory regions.
- Review your current secure application development processes. Ask yourself the following questions:
  - Are these processes sufficient to protect our data assets?
  - Are there any new threats present that were previously overlooked?
  - Are the proper hardware and software resources in place to protect our data assets?
- If you are activating and building a secure application for the first time, reach out to an expert(s) who will help guide you and ensure that it is done correctly.

These are just a few ideas to go a little bit further. Carve out time in your schedule each week to apply these action items. Even minor adjustments over a year can result in dramatic changes!

## Footnotes

1 https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/guide-to-iot-security.pdf

2 https://developer.arm.com/architectures/security-architectures/platform-security-architecture

3 https://en.wikipedia.org/wiki/PSA_Certified

4 https://community.arm.com/arm-community-blogs/b/internet-of-things-blog/posts/the-iot-architects-practical-guide-to-security

5 https://community.arm.com/iot/b/blog/posts/five-steps-to-successful-threat-modelling

6 A good general document to consult is the **NIST Guide for Conducting Risk Assessments**. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-30r1.pdf

7 That's right! Firmware is a system data asset! It's quite common for developers to overlook this fact.

8 www.alertmedia.com/blog/business-threat-assessment/

9 See Appendix A for the definition of "important terms."

10 Sometimes, authenticity is replaced with availability when we are considering robust software models.

11 www.mouser.com/pdfDocs/threat-based_analysis_method_for_iot_devices_-_cypress_and_arm.pdf

12 www.psacertified.org/development-resources/building-in-security/threat-models/

13 psacertified.org

14 https://bit.ly/3GPlIj3

15 If microcontroller selection dictated the architecture, then microcontroller selection would be Chapter 1, not Chapter 11!

16 www.beningo.com/insights/software-developers-guide-to-iot-security/

17 Just because we architect first doesn't mean that we can't in parallel experiment and evaluate different hardware solutions. Implementation feedback is almost always needed to evolve and tweak the architecture.

18 Definition is defined at https://bit.ly/3F4k5N2

19 https://arm-software.github.io/CMSIS_5/Zone/html/index.html

20 www.beningo.com/insights/software-developers-guide-to-iot-security/

21 There are also some interesting mechanisms for creating a hardware-based Root-of-Trust using a microcontroller SRAM known as SRAM PUF (physical unclonable function). SRAM PUF can be applied to any microcontrollers without the need for the microcontroller vendor to be involved. An introduction is beyond our scope, but you can learn more at www.embedded.com/basics-of-sram-puf-and-how-to-deploy-it-for-iot-security/

22 https://bit.ly/335RcTV

23 A list of labs can be found at www.psacertified.org/getting-certified/evaluation-labs/