arm

White Paper

Software Developers' Guide to IoT Security

October 2019

By Jacob Beningo, **President**
Beningo.com

Table of Contents

Introduction

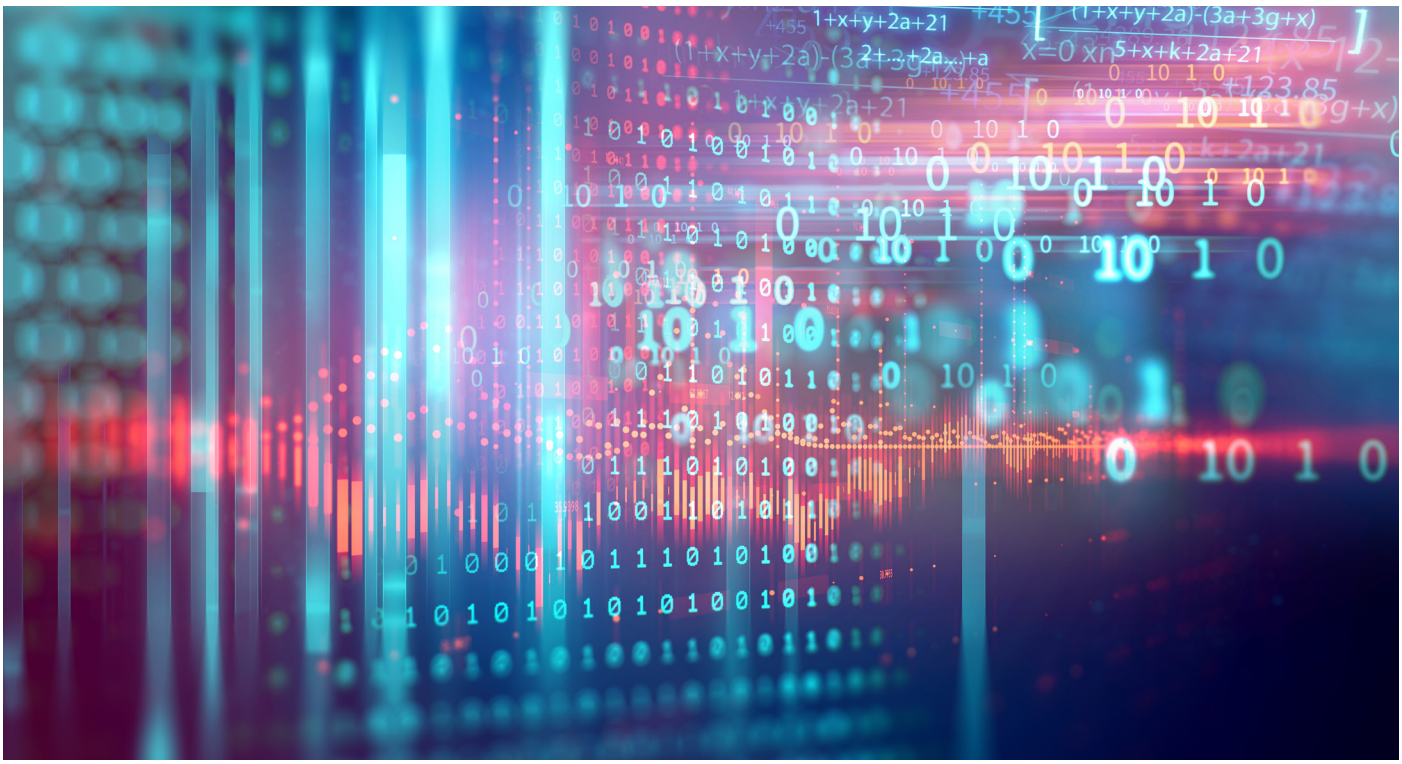
The number of devices connecting to the internet each year is growing at an exponential rate. In fact, Arm expects there to be over 1 trillion connected devices by 2035. While the functionality for so many devices can hardly be conceived now, the one thing that each device is going to need is a secure software implementation to protect it from the numerous threats. For many developers, though, security is an intimidating word. It's not completely clear to embedded systems developers what security is or how to implement it correctly. In this paper, we will explore what embedded software developers need to understand in order to develop secure IoT applications.

The Growing Need for IoT Security

For many companies, having security on their product seems to be a nice-to-have feature rather than a required feature. The idea that someone would try to attack, hack or exploit their product seems distant, improbable and complicated. Companies are often focused on getting their product to market with the features that are needed to support their customers, and security just doesn't feel like a priority in today's modern, fast-paced product development cycle. The fact though is that security isn't just nice to have, for any connected device it has become essential, a primary requirement and potentially even negligent to not consider security for a product.

Take for example several headlines from the recent past involving security violations with connected devices. The first, and probably most well-known among embedded developers, were hackers that found a way to remotely access Jeeps and control various settings while the car was in motion! The hackers were even able to control systems such as the engine, power steering and brakes (Video Link)[1].

Another example was a recent recall of nearly 500,000 pacemakers' due to holes in the firmware that allowed remote hackers to access and control the pacemaker[2]. In fact, security is overlooked and under-considered so often that a group of hackers recently created a killer application that could control an insulin pump because the FDA and the manufacturer didn't take their security vulnerabilities reports seriously![3]



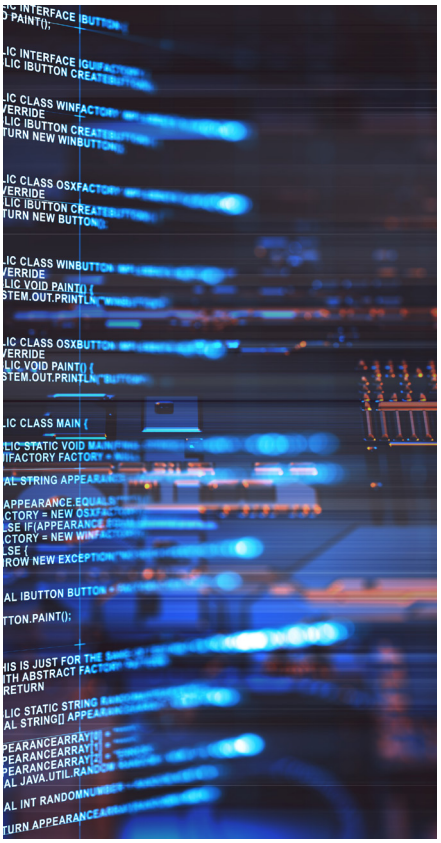
Security is no longer optional for four primary reasons:

1. The IoT now consists of billions of devices and while you might think the chances of someone finding your device is statistically improbable, just put a Linux machine on the internet with the default admin password and watch how the device doesn't last a day before it is discovered and hacked. (I know this will happen because I have actually tried it and the device didn't even make it through the weekend!).

2. Customers and businesses need integrity in their data to remain private and secure. Companies are developing applications that collect and store data about sleep patterns, heart rates, eating, exercise, travel and untold numbers of other data. It's imperative to users that their data will not be exploited, transmitted publicly or used against them in some other manner without their expressed consent. In today's world, we all know how valuable data is and if you can't protect that data, customers will go elsewhere to where they believe they will be protected.

3. Breached security of a device or its data can result in huge potential losses for the customer but, also for the device company. Developing and launching a product is expensive and often involves the development of numerous pieces of intellectual property, such as the source code. A device breach could expose years' worth of software development and allow hackers to exploit the software for their own purposes or sell trade secrets to competitors.

4. Security is no longer optional because the governments of the world are now stepping in and beginning to regulate the internet. Take for example the fine just issued to Facebook[4]. Hackers want to gain access to an IoT device and exploit its data in some manner and governments are starting to aim at technology companies that don't take security seriously. There are several examples of new regulations such as Europe's General Data Protection Regulation (GDPR), the IoT Cybersecurity Improvement Act of 2019 and the California SB-327.



What Needs to Be Protected?

As we recognize that security is now essential and must be included in every device, we need to stop and consider what exactly is it that we are protecting. It turns out that there are two general categories of assets:

1. General data assets associated with the device

These assets include things like the device firmware or software, the microcontrollers unique device ID and passwords for the user or device access. General assets can also be considered encryption keys that allow remote access to the device, cloud connectivity and manage secure communication.

2. Application-specific assets

These assets are directly related to what the product does. These might include sensor data like images, temperature, pressure and many other. Control data that manages to turn on LEDS, a pump, an actuator, etc. would also be considered a device-specific asset.

At this point, you might be wondering how do you go about securing an embedded system? How do you identify the assets to protect in your system, let alone actually put in place a mechanism to protect them? The best way to start securing your product in a cost and time-effective manner is to leverage an existing industry standard for security.

Accelerating Security with Platform Security Architecture

The Platform Security Architecture (PSA) was developed by Arm and ecosystem partners, to provide the embedded systems industry with a baseline security architecture that could be leveraged to secure an embedded product. PSA is designed to give you a collection of holistic resources to make security simpler: a set of threat model and security analyses documentation, hardware and firmware architecture specifications, an open source firmware reference implementation[5] and 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 both hardware and software. Take a moment to absorb that statement: security is not just something that is done in software, but also something in hardware! A secure solution requires both components working together.

It's important to realize that security is not just something that you bolt onto the end of your product when you are ready to ship it out the door. To get started writing secure software, you need to adopt industry best practices and start thinking about security from the very beginning of the project and ensure that the hardware can support the software needs.

Within PSA, there are four primary stages to develop a secure solution, which begin even before you start creating a product. This is because you need to identify the security needs, and let that dictate what is required in the hardware and software to secure the system.

There are four steps included in PSA that can be seen in Figure 1:

- Analyze
- Architect
- Implement
- Certify

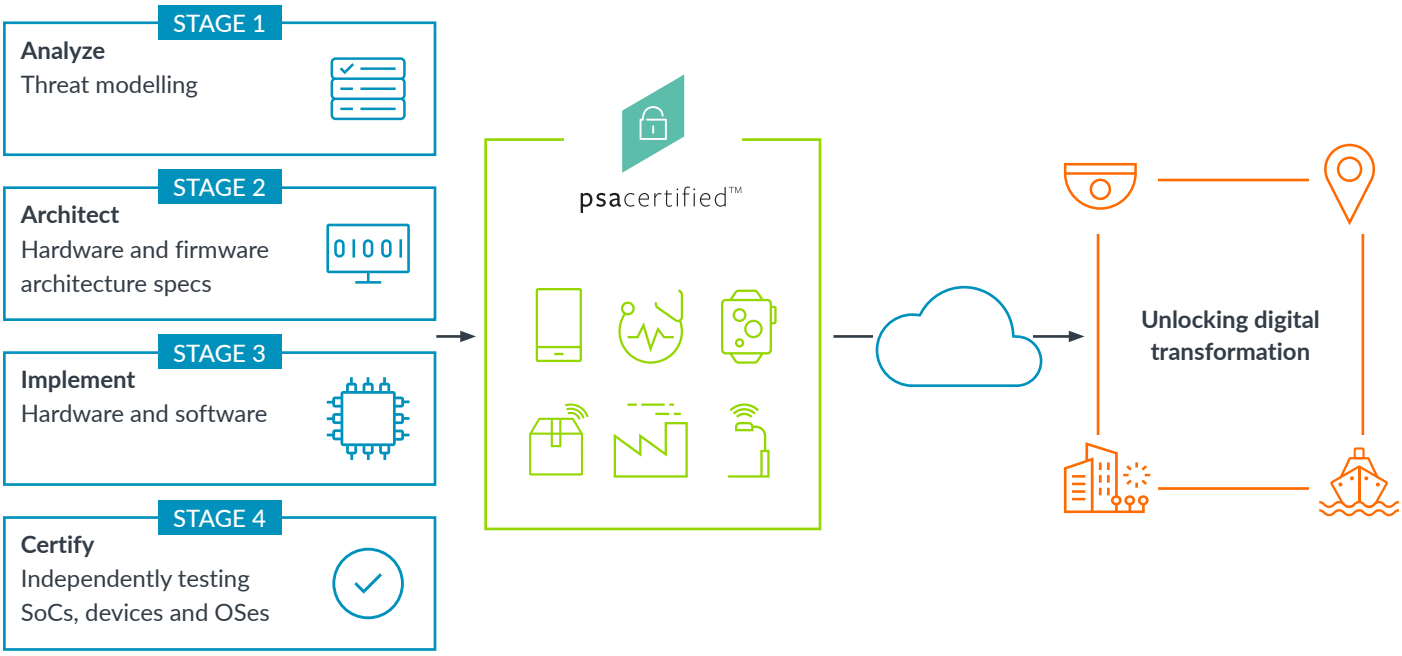


Figure 1 – The four primary steps of PSA

As you can see within the PSA model, there are two steps that come before the implementation that are critical to making sure the security of your systems is correct. Let's look at how to get started with secure software using PSA.

Analyzing a System for Threats and Vulnerabilities

Before selecting your microcontroller, you need to analyze your "to-be-built" product for threats and vulnerabilities. This involves identifying the data assets in the system and the threats that these assets will face. To understand the threats correctly, you need to define the security requirements for the system that will dictate not just the security strategies and tactics that will be used, but also the hardware and software that will be needed to protect those assets properly.

There are several different ways to analyze your system. One of the most simple and most comprehensive approaches is called threat model and security analyses, which is introduced in PSA. If you want to know more about the process, you can read all about it in this blog, or a joint white paper between Arm and Cypress Semiconductor.

Five Steps to Design Security Into Your Next IoT Device

There are five key steps that you need to walk through to analyze your system. These steps provide a comprehensive direction and example to follow to figure out how to protect your software assets.

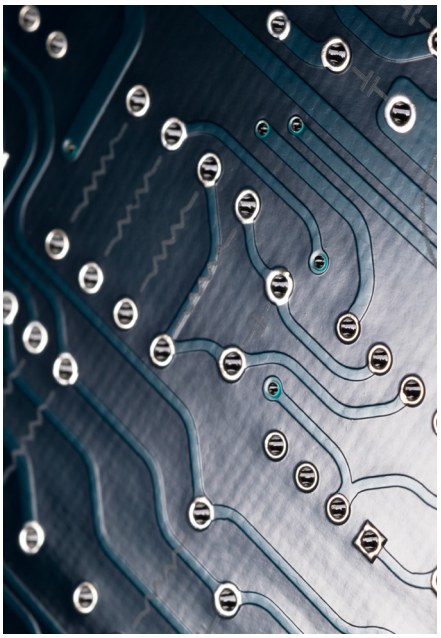
Figure 2 – The threat-based security analysis walks through the steps necessary to identify the data assets in the system, the threat types that those assets will face in the field, how to define the systems security objectives, and finally, how to generate requirements from those objectives that will protect the system's assets.

The outcome from following these steps and performing an analysis will be a list of data assets, the threats that they face and the security mechanism that can be used to protect the asset. When taken together, these can be used to create the requirements that dictate the minimum hardware set that will be necessary to secure the system. For example, common features that you'll find in the security requirements for a microcontroller will often include the following:

Feature: Hardware-based encryption
Solution: Encryption can be used to protect data

Feature: Digital signatures
Solution: Digital signatures can be used to verify the authenticity and integrity of firmware that is being loaded into memory

Feature: eFuse
Solution: eFuses can be used to lock down firmware features and create immutable ID's



Architecting a Secure Solution

The big question on a lot of developers minds when they have completed analyzing the system is how do I architect my security solution? It's important to understand not only what you are protecting and the mechanisms you can choose to protect them but, what's the best way to go about doing that? PSA outlines that using isolation within your microcontroller is key to securing your embedded system.

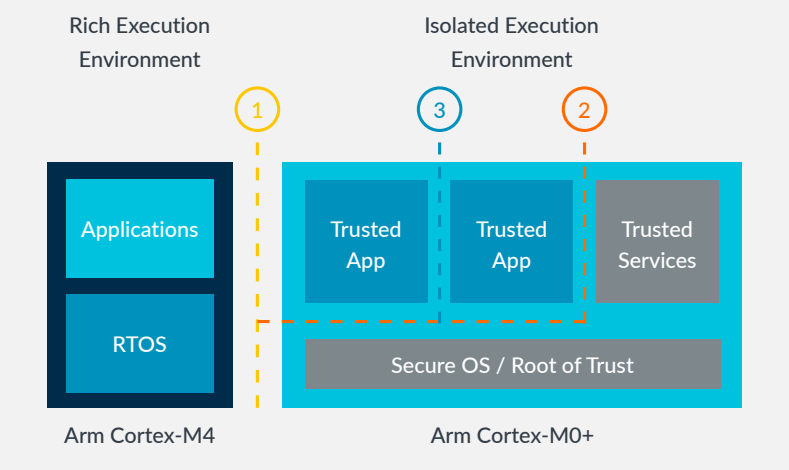
The best form of isolation that you can have in your system is hardware-based isolation, which allows you to separate the run-time environment into isolated regions that are protected from one another through hardware. On a single-core Armv7-M processor, the memory, peripherals and code are placed together into a single run-time environment that, once infiltrated, can be easy for an attacker to gain access to the whole system. The Armv7-M processors can run privileged and unprivileged modes along with leveraging an Memory Protection Unit (MPU) to protect processes, but these are still all executed "out in the open" on a single core.

To protect assets, you need to build hardware isolation to create separate, disparate run-time environments. By doing this, if an attacker was to gain access to one of the regions, they may be able to cause mischief in that one region, but would be unable to gain access to code and resources in the other regions due to the hardware isolation. For example, if a hacker were to gain access to a communications port, they still would not be able to access private keys and credentials that were hardware isolated in a separate hardware unit and protected by an MPU. Developers architecting their security solution need to account for various methods to provide isolation within their product.

There are two different methods you can use to isolate the run-time environment that don't involve using an external security processor:

Multi-core processors on a single chip

The multi-processor solution provides one CPU that is dedicated to running a trusted execution environment while the second CPU is designed to run general application code. The trusted execution environment is often referred to as the Secure code, while the general application code is Non-secure. For example, the Cypress PSoC 64 Secure Microcontrollers use a multi-core solution that has an Arm Cortex-M4 processor for its rich execution environment and running general application code (Non-secure environment), but also includes a Cortex-M0+ processor that runs all the Secure code (Secure environment). The secure processor runs the Root of Trust and trusted services completely isolated from the general processing core. An example of how the run-time looks can be seen in Figure 3, which also demonstrates how the isolated execution environment can also be further broken down into additional layers of isolation that you can utilize in the application.



Hardware-based isolation within PSoC 64 Secure MCUs enables secure element functionality and reduces the attack surface.

Three Levels of Isolation

1. Secure execution environment (SEE) isolated from rich execution environment
2. Root of Trust and trusted services isolation within SEE
3. Application isolation within SEE

Figure 3 – An example implementation of security through isolation leveraging multiple processors on the same chip in the Cypress PSoC 64 Secure Microcontroller. (Source: Cypress)

Single-core processor with hardware isolation

Arm TrustZone technology, included in the latest Armv8-M processors (Cortex-M23, Cortex-M33 and Cortex-M35P) provides hardware-enforced isolation within a single-core processor that separates the application run-time into Secure and Non-secure domains. The Secure domain is set up so you can have secure flash, RAM, peripherals and interrupts. The Secure domain can access all on-chip resources, while the Non-secure domain is unable to access resources allocated to the Secure domain. The switch between the Secure and Non-secure domains doesn't require any additional software during run-time. The CPU instructions to transition to the Secure zone are automatically inserted and are deterministic, requiring just two clock cycles! Figure 4 shows an example application that has Secure boot and Cryptographic libraries protected in the Secure Domain that then has a user application that accesses Secure domain functionality.

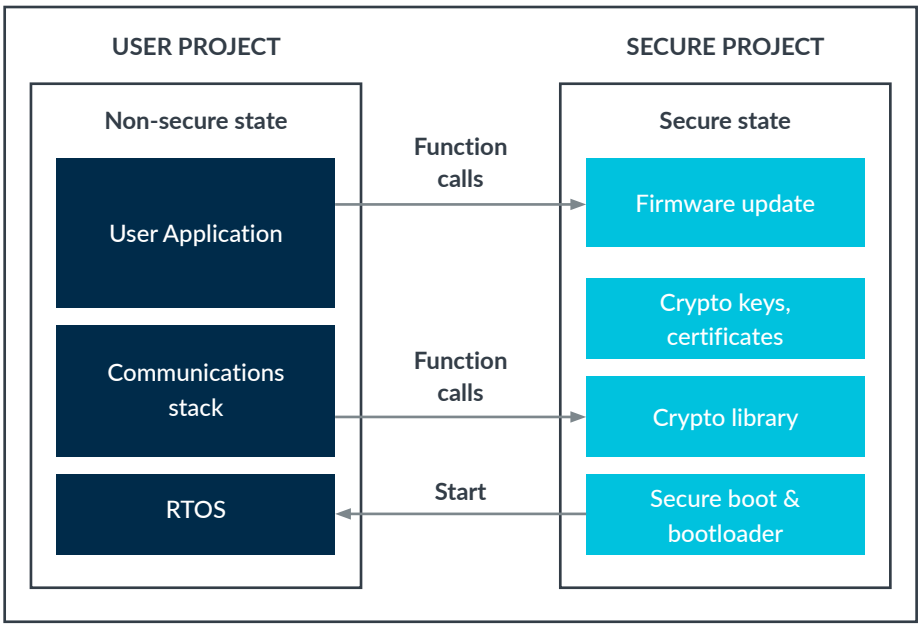


Figure 4 – A TrustZone application example

From Figure 4, it's important to note that when you are working on security solutions, there will no longer ever be a single-project application! Within the development environment, you will have two separate projects that represent the two isolated regions. This means you now need to decide where different applications belong, to the Non-secure user application or secure project. There may be a temptation at first to just put all the software into the secure project, but this completely defeats the purpose of having security through isolation. That's going back to a model for how to develop non-secure software.

Instead, you need to look at all the components that you are going to need in your application by creating a component diagram. From the component diagram, they can then start to partition the components between the Secure and Non-secure regions based on the function and whether they are being used to protect one of the assets that were identified in the analysis phase. If it is protecting an asset, or is an asset that needs to be protected, it should be placed into the Secure project, otherwise, it can go into the Non-secure project. Figure 5 below shows an example break down for how the components in an application can be separated into the new isolated paradigm.

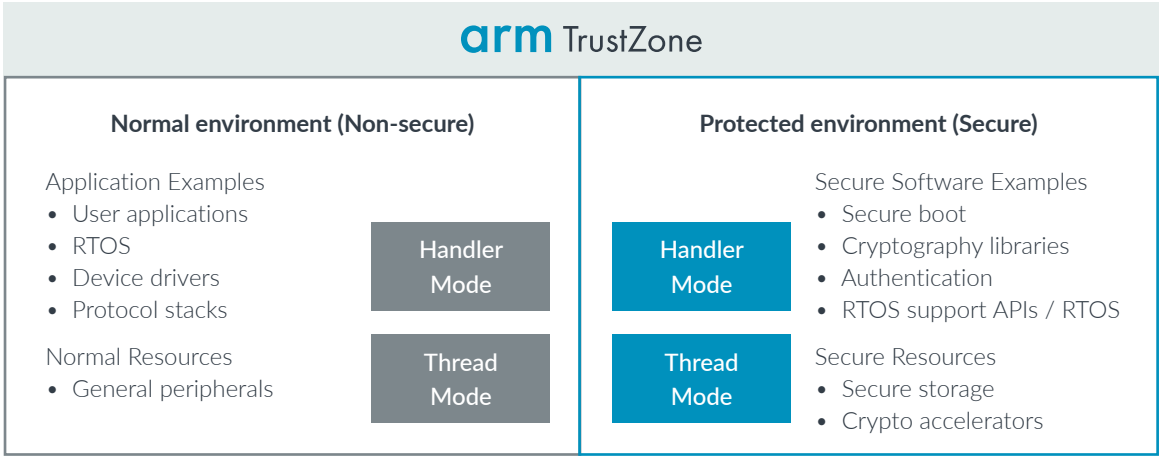


Figure 5 – Application components now need to be examined to determine if they are assets that need to be protected or are used to protect an asset. If they are, those components belong in the Secure execution environment. The remaining components can belong to the Non-secure application.

When you architect your security solution, it's important to realize that you aren't just looking at where different components and pieces of code and data should be located within the isolation memory regions. As developers, you may be using an RTOS which will also require you 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. As you architect your system, there are three different types of threads:

- **Non-secure only threads** - only calls functions that exist within the Non-secure isolation unit
- **Secure only threads** - only calls functions that exist within the Secure isolation unit
- **Hybrid threads** - executes functions within both the Secure and Non-secure isolation regions. These hybrid threads often need access to an asset that you are trying to protect

When you architect how your application should behave, you can use a UML Sequence Diagram to show the transitions in a hybrid thread between the Non-secure and Secure code. This can help clarify visually how an application will behave and what secure resources are being accessed. An example can be seen below in Figure 6.

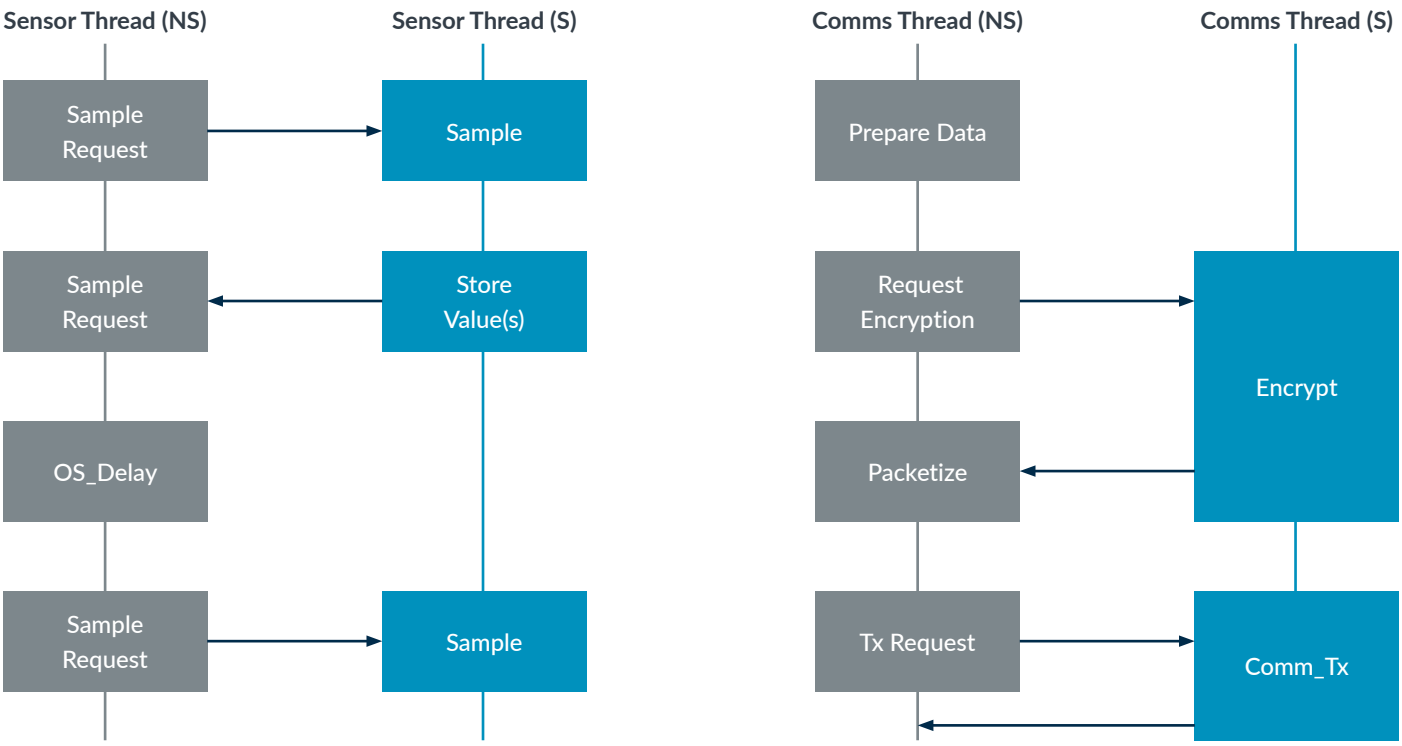


Figure 6 – Example sequence diagram for two application threads that have Secure and Non-secure function calls.

For more information, see the white paper "How should an RTOS work in a TrustZone for Armv8-M environment" by Joseph Yiu.

Secure Implementation

Once the secure solution has been architected, it's time to start the implementation. Before writing software, it's important to realize that there are several key components that can help accelerate software implementation. Let's examine each of these components and understand how they fit into the secure software implementation phase.

Secure Boot

The first component of implementation is the ability to securely boot the processes. A secure boot process will establish a Root of Trust by verifying the on-board chip certificate and booting the process in stages that verify the authenticity and integrity of all the firmware images that will be executed. Microcontroller vendors who are involved in security solutions will often ship their microcontroller with their **security certificate** that is tied to the microcontrollers unique ID. That certificate ownership can then be transferred to a manufacturing facility that will load the product firmware on the device or can be transferred to a developer that will use the microcontroller for development purposes. Once ownership has been transferred, you can then implement software, add keys, certificates and more.

The secure boot process should also verify the application integrity by calculating the **application hash** and comparing it to a known hash value. This can help ensure the application has not been unexpectedly modified. In a traditional embedded system, this would have been done using a cyclic redundancy check or a light-weight checksum that could easily be calculated by the firmware during start-up.

The last step in the secure boot process should verify the authenticity of the software that is going to be executed. Authenticity is often checked through the use of a **digital signature**. In a resource-constrained application, this can sometimes be a time-consuming activity, so the actual certificate may not be stored on the device, but instead a hash of the signature may be used.

Once the application has securely booted, there are any number of things that can be done. For example, the secure boot code could update any onboard security processes or simply jump to the application code to start executing the application. The exact steps at this point will be dependent upon the applications function and purpose. As a developer though, once secure boot is implemented, you need to investigate the existing software that you can leverage your design. This is where Trusted Firmware-M (TF-M) comes in.

Implementing Secure Firmware with TF-M

When implementing the product's firmware, it's important to note that you don't have to start developing it from scratch. There are software libraries that have been written with security in mind that can be used to accelerate your design.

For example, you can use Mbed, which includes several useful libraries such as encryption, MQTT and TLS, to name a few. If the design can be accelerated using existing libraries, this means there is also an opportunity to decrease development costs as well.

There is also a trusted open governance project that was started by Arm to create Trusted Firmware-M (TF-M). TF-M is a trusted code reference that provides examples on how to implement Secure code on a Cortex-M platform.

TF-M is built as a set of highly configurable software components suitable for constrained systems. It consists of secure boot and a set of secure runtime services including: secure storage, cryptography, audit logs and provisioning that can be used by applications[7].

When getting started with secure software for your embedded system, it is always highly recommended to review implementation examples from the microcontroller vendor that will be used with the product:

- Security application examples
- Reference Secure code
- Identifying additional hardware and software features that are outside the processing core
- Security use cases

For example, if you are developing an application that uses the Microchip SAML11, a TrustZone enabled microcontroller, you will find several example projects that demonstrate use cases such as protecting software IP, developing secure sensor applications and how to protect and recover a system from malicious attack attempts. (Example application can be seen below in Figure 7).

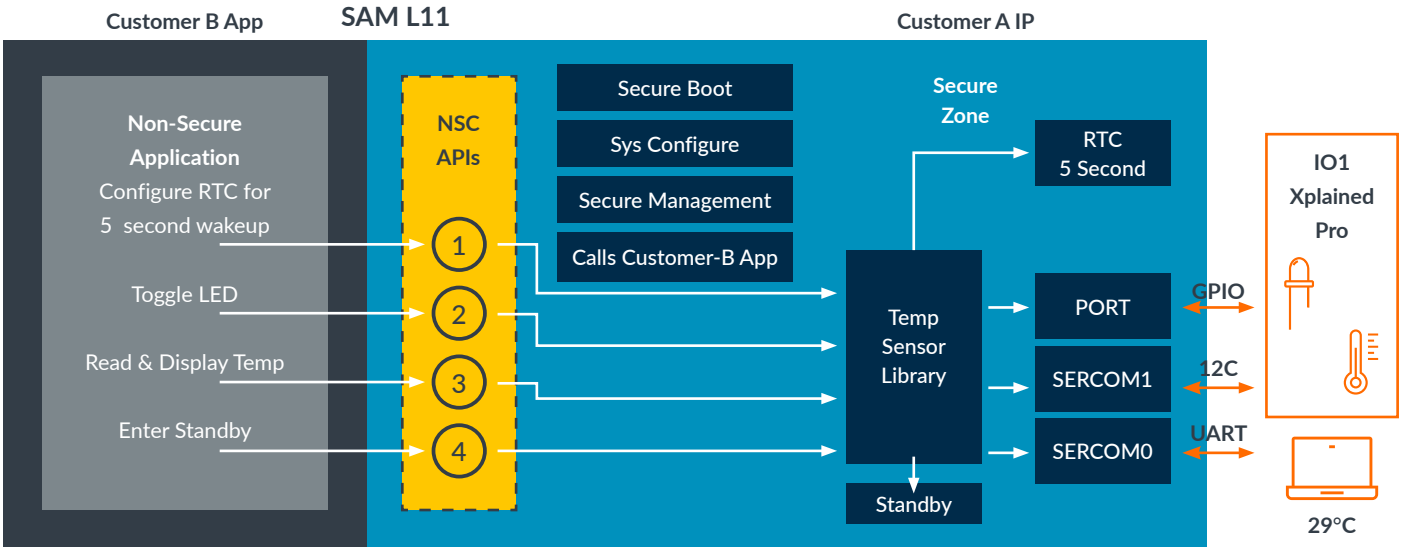


Figure 7 – An example application from Microchip using the TrustZone enabled SAM L11 demonstrates how you can partition the application for Secure and Non-secure run-time environments.

You will often find that between all these different sources, you will be able to leverage existing security materials and secure your embedded system.

Implementing Software Building Blocks with CMSIS

Many Cortex-M developers have become familiar with leveraging Cortex Microcontroller Software Interface Standard (CMSIS) in software designs. CMSIS is a vendor-independent hardware abstraction layer for devices that are based on Arm Cortex-M processors. CMSIS allows you to mix and match software components from multiple software vendors that best meet your software needs. CMSIS has been used by developers for more than 10 years now and supports more than 6,000 devices. Vendors who support CMSIS often provide software through CMSIS packs, which are open source and available on GitHub.

You might think that secure software would not be supported within CMSIS, but that thinking couldn't be further from the truth. Since June 2019, CMSIS includes TF-M with a generic hardware abstraction layer that was adopted to support Cortex-M23 and Cortex-M33 based devices. This means that you can leverage CMSIS and TF-M to dramatically simplify software configuration for custom hardware. As we discussed earlier, a key ingredient to successfully implement security is to partition and isolate your software which is now available through CMSIS-Zone[10].

Partitioning Designs with CMSIS-Zone

Security has added complexity to the software development lifecycle, particularly the implementation phase. Using hardware isolation such as TrustZone now requires you to manage memory, peripheral and other system resources across multiple software projects and sometimes even across multiple processors. Managing these resources easily in one place is what CMSIS-Zone is designed to do.

CMSIS-Zone includes an Eclipse-based utility that provides a simple GUI to assign resources.

The utility[8]:

- Displays all available system resources including memory and peripherals
- Allows you to partition memory and assign resources to sub-systems
- Supports the setup of Secure, Non-secure, and MPU protected execution zones with the assignment of memory, peripherals, and interrupts
- Provides a data model to generate configuration files for tool and hardware setup

The CMSIS-Zone utility makes it easy to simply click and configure the system instead of needing to manage configurations in each project. An example screenshot for the open-source utility can be seen in Figure 8.

Other development tools use the output of CMSIS-Zone to set-up Secure and Non-secure projects in a consistent way. Software developers of these projects can rely on the configuration and create the application based on it.

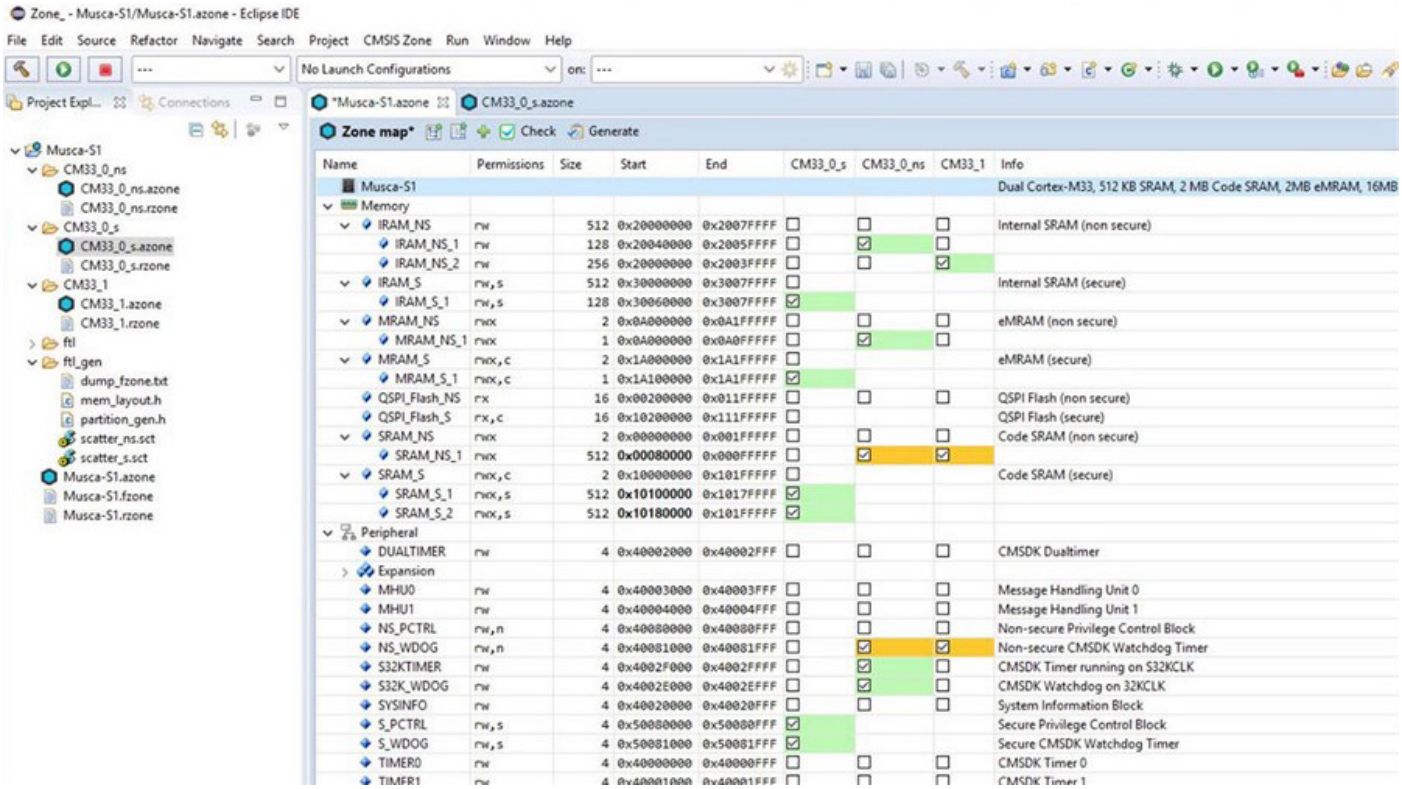


Figure 8 – The CMSIS-Zone configuration allows a developer to partition their system resources across multiple projects and processors (Image source: Arm8).

Secure Cloud Connectivity with the Pelion IoT Platform

The last piece of the solution is for providing secure credentials and updating devices at scale. Pelion IoT Platform helps to do so by providing the infrastructure to securely manage and connect your device to the cloud.

The Pelion IoT Platform is best described as a secure and flexible foundation of IoT services for connectivity, device and data management[11]. Pelion abstracts the complexities associated with device and data management that are often associated with IoT devices and solutions. Pelion provides several key features to accelerate your solution:

1. Secure management capabilities

You can securely manage your devices to onboard them when they are deployed in the field, enable secure communication with your preferred cloud vendor and even manage firmware updates. You are even able to manage the full lifecycle of the device from onboarding to decommissioning.

2. Easily manage data

With the massive amounts of data generated by IoT devices, you need to be able to sift through that data to decrease time and the cost to utilize the data and maximize the companies' opportunities.

3. Connect across the world

Devices across multiple networks around the globe can all be managed within a single environment. You can manage important connectivity issues such as auto-deployments, device operations and of course billing.

Using TF-M, CMSIS, CMSIS-Zone and Pelion will allow you to quickly and efficiently get your security architecture implemented on your TrustZone enabled microcontroller.

Certifying That a System is Secure

Once the software is implemented, it's possible to certify both hardware and software to the PSA in order to provide developers, but also consumers, assurance that industry best practices have been followed when implementing the product.

The fourth step of PSA, PSA Certified, is an independent security evaluation scheme for IoT chips, OSes and devices. It provides a much-needed scheme for constrained devices, plus a common language that the whole industry can use and understand.

There are three levels to PSA Certified, which each increase the security robustness that an IoT device might need:

psacertified™ level one	 psacertified™ level two	 psacertified™ level three
PSA Certified Level 1 (a document and declare with lab check) covers the foundational security requirements, considering the PSA Security Model goals, plus government requirements.	**PSA Certified Level 2** steps things up to mid-level assurance and robustness with time-limited white box testing.	**PSA Certified Level 3** covers more substantial, extensive attacks, such as side-channel and perturbation.

Conclusion

Security is no longer optional for connected devices. Implementing a security solution can seem intimidating at first, but there are numerous industry best practices and standards that you can use when developing a secure product. To be successful in such an endeavor, you need to follow an existing architecture that has not just the steps but also the ecosystem to help support. When following the steps laid out by PSA, security is simplified by allowing them to focus on the assets they need to protect and the threats those assets face. The mantra of "security through isolation" can be adopted, which will then lead you to focus your architecture on separating the code into secure and non-secure code segments.

To get started developing secure software for your device, explore the resources below:

TrustZone

- TrustZone webpage

PSA

- PSA webpage
- Whitepaper: Platform Security Architecture
- Webinar: Platform Security Architecture
- Arm's guide to threat modelling
- Other PSA resources: www.arm.com/psa-resources

CMSIS

- CMSIS webpage
- Blog: CMSIS: A success story
- Blog: What are CMSIS software components?
- Blog: Validating your IoT system with PSA and MDK
- Blog: Configuring Armv8-M systems with CMSIS-Zone

Trusted Firmware-M

- TF-M webpage

References

1) https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/
2) https://www.theguardian.com/technology/2017/aug/31/hacking-risk-recall-pacemakers-patient-death-fears-fda-firmware-update
3) https://www.wired.com/story/medtronic-insulin-pump-hack-app/
4) https://www.cnn.com/2019/07/24/tech/facebook-ftc-settlement/index.html
5) https://developer.arm.com/architectures/security-architectures/platform-security-architecture
6) www.cypress.com/psoc6security
7) https://developer.arm.com/tools-and-software/open-source-software/firmware/trusted-firmware
8) https://arm-software.github.io/CMSIS_5/General/html/index.html
9) https://community.arm.com/developer/tools-software/tools/b/tools-software-ides-blog/posts/configuring-armv8-m-systems-with-cmsis-zone
10) https://arm-software.github.io/CMSIS-Zone/index.html
11) https://www.arm.com/products/iot/pelion-iot-platform

arm