Electronic Design

By KIMBERLY DINSMORE, Security Solutions Manager,
Renesas Electronics, www.renesas.com/iot-security

Top Strategies to Improve Microcontroller Security (Part 2)

From isolation measures to secure and first-stage bootloaders to DOTF, engineers can opt for a range of solutions to better safeguard their MCUs from outside threats.

Strong security requires a strong foundation. Before the MCU can secure the entire system, it must secure itself. While it's vital to determine the relevant threat model and security policy for a specific product, some security solutions are very broadly applicable. Part 1 looked at a couple of the simpler solutions. Part 2 delves into other, more aggressive steps engineers can take to secure MCUs.

Keeping Things Separate: Isolation Mechanisms

The concept of isolation is new to microcontrollers. Since MCU code is often developed by a small team or even by a single engineer, it may seem pointless to protect the MCU from itself. But now that MCUs are active on the internet and applications are created from a significant amount of third-party code, that level of protection needs to be considered. Isolation mechanisms can also protect against inadvertent damage from run-away code, potentially serving as a safety feature.

The best protection mechanisms are hardware-enforced isolation. Memory protection units (MPUs) can be very effective, but they often have coverage gaps. Proprietary mechanisms such as Arm TrustZone fill many of these gaps.

One thing to watch for: How are the boundaries between the trusted and the non-trusted regions set? If they're set in software, then malicious code could potentially alter them. The strongest solutions set these boundaries immutably, but this introduces the potential complication of requiring firmware updates to remain within a specific size.

Some questions to ask include:

- What third-party code should potentially be isolated?
- What code is so complex that it's impossible to provide 100% test coverage?
- What assets and services do you need to protect from inadvertent or malicious alteration?

Protecting Firmware Updates: Secure Bootloader

Your product may not be as high-profile as a smartphone, but even humble devices such as connected thermostats can offer hackers a way into a targeted infrastructure. As a result, security consortiums and governments worldwide are recognizing the need to be able to update firmware in connected devices to protect against security threats. To ensure that this capability doesn't introduce more problems, this update must be done securely.

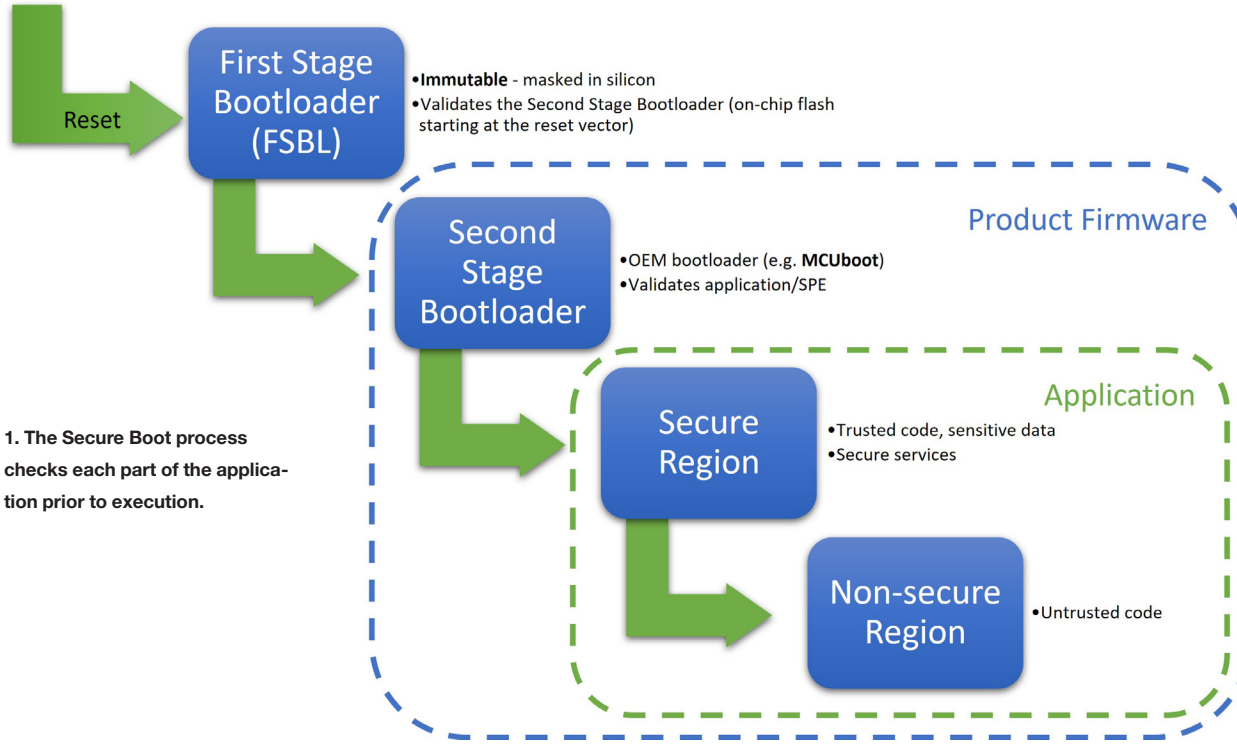
There are multiple ways to create a secure bootloader, and as tempting as it is to create one from scratch, it's much more efficient to leverage a trusted open-source project. Many MCU vendors even offer ports of these solutions in their software packages. Such solutions are configurable for most common use cases, and since they're open source, you can modify them for your specific needs.

The best thing about these solutions is that you're leveraging years of "lessons learned" from experienced firmware engineers. And the implementation is subject to inspection and review by security experts to ensure there are no hidden "back doors."

Since the requirement for supporting firmware updates is going to be required by government legislation, the questions here are more limited. Simply, do you have an external interface over which a new firmware image can be received?

Taking Cues from Microprocessors: First-Stage Bootloader

The secure bootloader solution mentioned earlier typically focuses on ensuring that only authentic firmware updates



1. The Secure Boot process checks each part of the application prior to execution.

are accepted. Many of these solutions also offer another feature—the ability to authenticate the application code prior to executing it. But what ensures that the secure bootloader itself hasn't been corrupted?

Microprocessors solve this dilemma by using a first-stage bootloader. It's a small amount of executable code built into the silicon that authenticates the application's secure boot solution, which is then termed as the second-stage bootloader.

Figure 1 illustrates the secure-boot process when an isolation mechanism is also included. It's good to note that these bootloaders can double as a safety feature, ensuring that the code hasn't inadvertently been corrupted. One thing to watch for: It will take some time to perform an authenticity check on the code, so study the available options to find the best balance for your product.

Some questions to ask include:

- Do you need to authenticate or integrity check all, or a portion of, application code prior to execution?
- Will you ever want to update your second-stage bootloader?
- Do you have any start-up times you need to meet?

The Keys to the Kingdom: Secure Key Storage

Similar to securing entry to a building, securing an embedded application often requires the use of keys. Cryptographic rather than physical, the format might be different, but both the concept and consequences of careless handling are fundamentally identical.

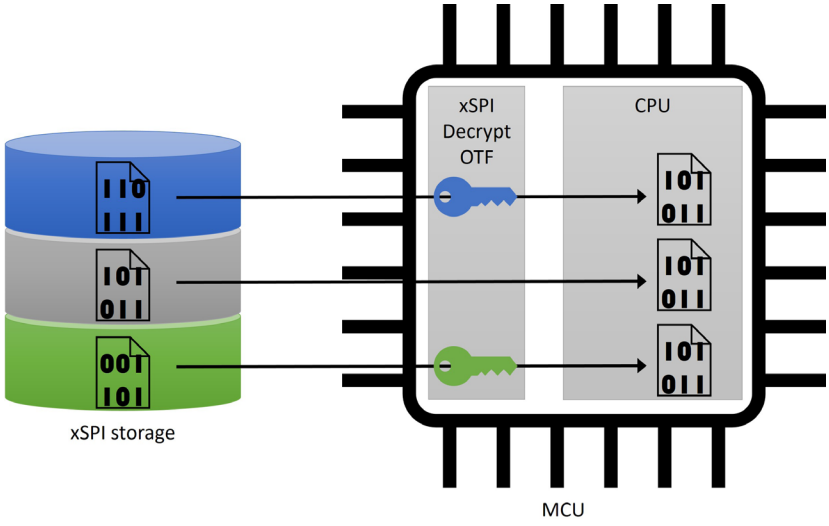
Lose your keys and you can no longer get in. If someone else finds them, they now have complete access. If they copy your keys, they can do significant damage before you're aware that something is wrong. This security feature is so critical that some industries require that the key storage mechanism have specific certifications.

Keys can be compromised in a variety of ways. If an attacker has only remote access, then keys stored or used in plaintext are vulnerable if malicious code can be injected. If an attacker has physical access, plaintext keys are vulnerable if the external interface protection is compromised. There are also more sophisticated attacks, such as analyzing the electromagnetic emissions of the device during a cryptographic operation or measuring the time required to complete a cryptographic operation.

Since these attacks are non-invasive, only inferring information from observed behavior, they're called side-channel attacks. In addition, attacks can be performed by causing the cryptographic operation to fail and analyzing the partial results. Since these attacks require altering the operational environment of the device, they're called fault injection attacks. Cryptographic keys are one of the most critical elements of a security solution, making them a prime target for attack.

Questions to ask here include:

- Do you require key storage with a specific security certification? Due to the nature of FIPS and Common Criteria certifications, such solutions tend to require an external component. These often have the disadvantages of increased cost, increased power consumption, slower execution time, and increased development complexity. There's a small number of certified microcontrollers with integrated secure key storage, so be sure to check



2. Decryption on-the-fly provides confidential storage of code and data external to the microcontroller with minimal impact to execution speed.

out those. Certifications like PSA Certified and SESIP offer alternatives to external secure key storage, but be sure to look at the certification details to see exactly what you're getting.

- Are physical attacks within scope? If so, avoid any solution where keys are stored or used in plaintext. Hardware cryptographic implementations with side-channel protections are useful here.
- Are remote attacks within scope? If so, again, try to avoid solutions where keys are stored or used in plaintext, but minimally, pick a solution that incorporates an isolation mechanism that can offer some protection.

Safeguarding Code and Data: DOTF

Unlike microprocessors, microcontrollers typically keep all of their code and data inside the chip itself. Some applications, though, require the processing power of an MCU but with more code or data than the MCU can store.

Most MCUs can use serial interfaces, such as SPI or I2C, to interact with an external flash device, which can be preprogrammed with data used by the application or programmed with data generated by the application. Some MCUs can even execute code directly from these external flash devices. But now we have a new security consideration: How can we protect the confidentiality of this external code and/or data?

Decryption on-the-fly (DOTF) is designed for seamless reading and/or execution of encrypted external data. There will be a time penalty over reading/executing plaintext data/code. However, it's insignificant compared to "manually" determining the required block, decrypting it into a temporary buffer, and reading/executing from that buffer.

Application development is also simplified, especially for the case of executable code, since the code is simply compiled to execute at the address where it's stored in the external storage device. As shown in *Figure 2*, some DOTF solutions allow for different keys to be used over different ranges, including areas of no encryption.

Questions here include:

- Will you be using an external device to store code or external data?
- Do you need to maintain the confidentiality of that code/data?
- What are the ramifications of a third party obtaining that code/data?

Confidential Communication: Secure Internet Connection

When the first internet-connected 8-bit microcontrollers were a novelty, no one really thought much about the security of the connection. Back then, these devices simply weren't a target for attack, and the concept of exploit chains was mostly unknown.

More importantly for the engineers developing these microcontroller solutions, it was critical that the IP stack was lightweight and efficient. Adding a Transport Layer Security (TLS) layer, besides being complicated, produced code that was neither.

The good thing is that now, most developers understand and accept the need for a security layer such as TLS in their communication path. Most public cloud services require it in the form of TLS or DTLS. The only question here is: Do you need TLS or DTLS? The answer will depend on your specific communication protocol (TCP-based or UDP-based).

Striking the Balance Between Perceived and Actual Threats

"Security" has many definitions, with a wide range of threats and varying severity of consequences. The mitigations for these threats often have a profound impact on the architecture of a product. It's vital to decide up-front what assets need to be protected and how you intend to protect those assets from the identified threats. Only then can you create a proper requirements specification and architecture design for your product.

Some of the most successful hacking techniques, though, don't involve technology. Social engineering attacks can trick people into divulging sensitive information. Bribes can provide legitimate access to illegitimate users. As engineers, we need to provide a solution that attempts to find the balance between the perceived and the actual threats, ensuring that the technology isn't the weakest link.

Kimberly Dinsmore is the Security Solutions Manager for Arm-core microcontrollers at Renesas Electronics.