SECURE EMBEDDED SYSTEMS: A DEVELOPER'S JOURNEY

SHAWN PRESTRIDGE, IAR SYSTEMS

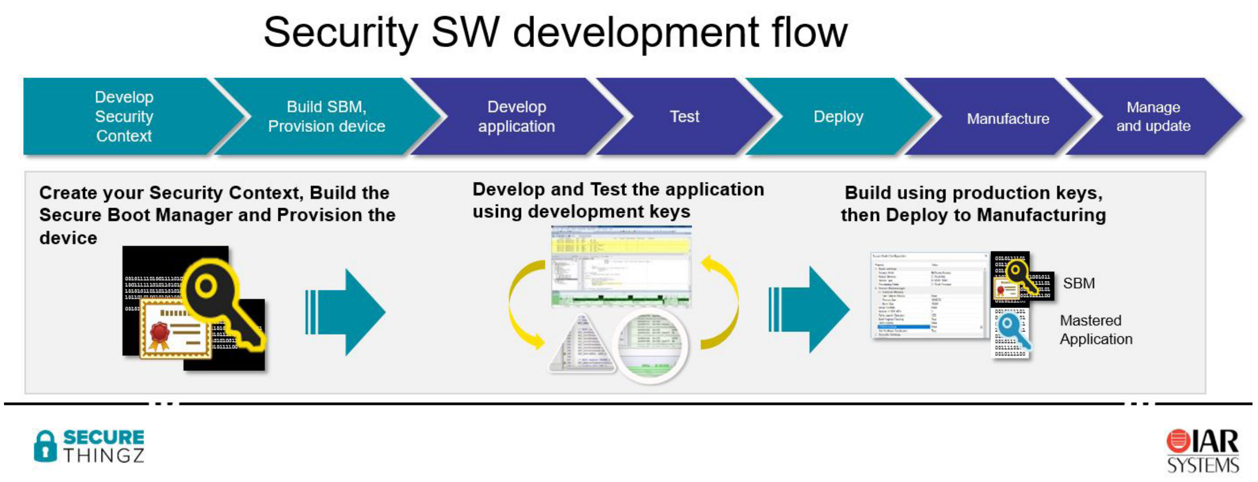
Properly implemented security starts at the beginning of the development process. And continues throughout the entire design cycle, all the way to manufacture.

Overview

Security is a significant problem in the embedded space, particularly in the industrial, automation, and automotive sectors. There are a host of reasons for these security lapses. However, many could have been avoided had the developer started his design on the correct path right from the start. Attempting to fix security issues later, especially after a product has shipped, is not recommended.

If we make security part of the developer's DNA, many issues can be avoided. But what does that mean? It means that security must be "front of mind" through *every* phase of the design process—right from product/idea inception. Such thinking requires that the developer check all the appropriate boxes along the way. Those layers or stages the developer will go through include:

- HW Root of Trust (RoT) and Secure Boot Services
- RTOS and Core Services
- Communications and Data Security
- Applications Security
- Authentication and Authorization


Security SW development flow
Develop Security Context
Build SBM, Provision device
Develop application
Test
Deploy
Manufacture
Manage and update
Create your Security Context, Build the Secure Boot Manager and Provision the device
Develop and Test the application using development keys
Build using production keys, then Deploy to Manufacturing
SBM
Mastered Application
SECURE THINGZ
IAR SYSTEMS

The Starting Point: Who and Where

Before laying pencil to paper, so to speak, developers must understand the legislative issues associated with the deployment geography of their end products. In other words, the region in which the end product will be deployed could change how security is deployed, because different regions have different rules when it comes to security.

Next is to determine the profile of the typical user and the use cases the end product may be subjected to. Where and how often will the product connect to the Internet, and what other people and/or devices have similar access? How will updates occur? It may not be possible to always know the answers to these questions for all products, as use cases change over time. For that reason, developers should always err on the more conservative side, and assume the worst-case scenario.

One great starting point is a framework that has been laid out by the IoT Security Foundation, whose charter is to secure the IoT by composing and maintaining a comprehensive compliance framework of recommended steps for the creation of secure IoT products and services. That framework consists of a 13-step program of best practices that serve as a guide for developers. Most developers won't follow all 13, but the more you can do, the more secure your system will be.

IP Protection

Your IP may be the most important thing that your company owns. If someone were to steal your IP, they could recreate your products or some portion of your products or the functionality that goes along with it. Or, a potentially worse scenario could be that your IP is altered within a device. Effects from that situation could be devastating, from both liability and branding perspectives.

It could potentially be even more difficult to protect your IP when your device is connected to the IoT, either with a wire or wirelessly. In this scenario, you may not have control over what's plugged into the network, and as the saying goes, "you are only as strong as your weakest link."

A big part of IP protection is protecting your application. That could be partly in hardware and it could be partly in software/firmware. In either case, the first step to security (and you'll hear this again and again) is creating high-quality, secure code right from the beginning.

One way to make that happen is to take advantage of IAR System's C-Trust security tool. C-Trust works hand-in-hand with the company's popular Embedded Workbench tool suite. When you start your project, you enable the security setting within the tool, which adds the encryption keys and denies version rollbacks. The image that's generated is encrypted, and that's the majority of what the developer needs to do, at least at this point.

Secure Contexts

A primary goal of secure contexts is to prevent attackers from accessing the APIs of that industrial platform; again, extra care should be taken for IoT platforms. In doing this, the contexts capture all aspects of the system, especially the parts that need to be locked down. They also represent the minimum standards of authentication and confidentiality.

Keeping with the popular theme, those secure contexts are inserted at the initial design stages. In theory at least, that should keep the system secure all the way through to manufacture, with the end product being produced exactly as the manufacturer intended.

The secure context defines the trusted execution environment's complete configuration. With C-Trust, developers can quickly apply security contexts to application development to ensure consistency throughout the product development.

Specifically, C-Trusts simplifies:

- complex cryptographic device identities and ownership structures
- creation of a root-of-trust, ensuring device authentication, authorization, and attestation
- secure boot manager extensions
- application of patches and updates
- identity delegation for onboarding into cloud infrastructure applications

If implemented properly, these security contexts provide the foundation for a secure platform.

Secure Boot

Just as we've discussed that your design must start off on the right (secure) foot, that's also the case for when your device boots up. If the booting process is not secure, you can assume that your entire design looks "unlocked" to the outside world.

The literal definition of secure boot is that it's a mechanism for ensuring the integrity of firmware and software running on a computing platform. It's vital that the platform guard against malicious attacks or even unauthorized software updates that could happen prior to the operating system launching.

During the development process, an authentication key is created. To enable a secure boot, the firmware is signed with that key and verified in the end product. Each time the device boots up, it verifies the firmware signature using the matching key.

The secure boot manager, which is responsible for the booting process, acts like a "trust anchor." Note that, during the boot process, the importance of running secure code cannot be emphasized enough. That secure boot manager must run every time the system boots and should never be altered. It has the important job of ensuring that you're always running legitimate code. Whenever firmware updates come in, it's up to the boot manager to ensure that they are legitimate and authentically signed.

The process of developing a secure boot manager is integrated into the C-Trust security tool. During development, the tool will provide low-level device services to manage access to all devices.

Secure Provisioning

Secure provisioning helps ensure that the software released by the development team is exactly the same as the software that's installed in the end devices. Any changes that occur at this point can be detrimental to the functionality of the device. And it's difficult to know if those changes came from a verified source. Such protection is needed for brand protection and for the safety and security of end users.

While we often just think of software provisioning in terms of the deployment part of software delivery, it's actually much more than that. Secure software provisioning also helps manufacturers quickly and reliably manage the various aspects of the platform, which could include information about systems, users, and applications. The full provisioning process could also include parts of the test, quality assurance, and reporting stages.

The provisioning can be an arduous task, as it encompasses device configuration, including identity and key provisioning. However, C-Trust greatly simplifies that process. Through a semi-automated process, the tool securely transfers the complete design to a production environment, first for prototyping and later for mass production. In the transition, the secure development keys are replaced by secure production keys, during the creation of a final, secure production package. C-Trust automatically imports these production packages.

Conclusion

The bottom line is that security needs to be prioritized to the top of the developer's checklist. In fact, it must rule the list, and be so ingrained into a developer's DNA, that it's something that he/she just does without thinking that it's an extra step. The best way to do that is to make it as easy as possible, and remove all of the guesswork for the developer.

That's where C-Trust shines. As stated earlier, it's a security development tool that works as an extension to IAR's popular Embedded Workbench tool suite. Hence, you operate in an environment in which you're already very familiar. The result is the delivery of secure, signed, and encrypted code with pre-baked security context profiles for IP protection and production control.

In addition to the Embedded Workbench integration, C-Trust offers support for most of the popular microcontrollers, including those from ST, Renesas, Microchip, and NXP. And new devices are continually being added. So there's no excuse not to be fully compliant. Make security part of the developers' DNA. Security from inception.

For questions, please email fae@iar.com