Floodgate Security Framework
Secure Boot for Renesas Synergy Boards

The Floodgate Security Framework includes a Secure Boot facility enabling OEMs to deliver devices utilizing signed code to impose varying levels of trust on runtime applications. Services are delivered with an extensible framework that is scaled-down for resource-constrained systems while offering higher degrees of certainty for newer systems with hardware security support. This results in a consistent Secure Boot solution available for a large number of vendors and processor families.

Overview

A secure boot process seeks to ensure only authorized firmware executes on a device by using code signing to create a cryptographically signed image which is loaded onto the device. The image is validated by the secure boot process integrated into the bootloader on the device. This ensures only authenticated code signed by the OEM is allowed to run on the device.

Secure boot protects against attempts to reprogram the device with malicious firmware or that attempt to modify the firmware running on the device.

While most cyber-attacks target PCs, tablets and mobile phones, the growing success of embedded (IoT) solutions has reached the point numbers where they have become a lucrative target. With field device lifetimes that span decades, this threat is must be addressed now.

The Secure Boot facility provides risk mitigation for the following:

- Malicious modification or sabotage of system components or configuration data
- Execution of untrusted components and services from compromised updates
- Protection from device misconfiguration leading to failure

In its most minimal form, the Secure Boot facility:

- Allows system developers to securely sign application code and data before execution
- On system startup, verifies the integrity of code and data before permitting execution
- Provides a secure mechanism for identifying individual devices
- Stores a secure audit log for system boot processing and additional services
- Offers a way to revoke and replace compromised cryptographic primitives

UEFI Secure Boot

Over 140 companies participated in the UEFI Consortium to replace hte PC BIOS with a unified framework (Unified Extensible Firmware Interface). Standardization includes UEFI Secure Boot which aims to ensure a device loads the target OS free from pre-boot malware (rootkits). Floodgate Secure Boot differs from UEFI and focuses on the requirements of embedded systems.
Hardware Support
Many embedded systems include hardware support for security including cryptographic acceleration, secure key storage, policy-based access control for critical registers, and sometimes provisions for secure startup. Floodgate Secure Boot services utilize these capabilities when present. In systems that do not have hardware support, the Floodgate Security Framework provides the required software components including encryption and hashing algorithms, random number generation and secure key storage.

Secure boot process
Secure boot uses code signing to verify the firmware before allowing the firmware to run. Coding signing is achieved by calculating a SHA hash of the firmware image and signing that hash string using RSA signatures. A short description of this process is provided in the code signing fundamentals section of this document. Code signing produces a signature for the code image that is loaded into the embedded device, along with the signature. The device must be programmed with the public key corresponding to the private key used to sign the code. Programming of the private key would normally occur in the factory when the device is programmed. The code image and signature can be updated in the field, provided it is signed with the same private key. Normally, the public and private keys are stored in a certificate.

On the embedded device, secure boot works by validating the firmware on the device matches the signature. This occurs during the normal device boot process and the device will only be allowed to boot and run if the firmware is valid.

The secure boot loader first calculates the hash of the firmware loaded on the device. Next, it uses the stored public key to decrypt the stored signature that is supposed to correspond to the firmware image. The decrypted signature is the hash value of the firmware authorized to run on the device. If the two hash values match the firmware about to be executed is the same firmware that was signed by the code signing application. We also know it was signed using the private key that corresponds to the public key stored on the embedded device. The use of certificates to store signing keys allows the keys to be associated with the OEM that developed the device.
Code signing fundamentals
This section provides additional details on the cryptographic methods used to provide the basic features of Secure Boot. This text assumes a basic understanding of cryptography.

Data Signing
Data can be signed with a cryptographic key such that non-signing parties can follow a process to verify the integrity of the signature and associated data. This is done using asymmetric cryptography, which utilizes two mathematically related cryptographic keys. One key – the private key – is used to sign a chunk of data while another key, the public key, is used to verify the signature. Verification ensures that the information contained within the signed message has not changed since it was signed, and virtually guarantees the signature was done with the public keys’ private counterpart.

Certificates
If you can ensure a private key is only known to a specific person, group of people, or electronic resource, signing can be used to ensure that data is sourced by that person, group, or resource. In order to achieve this end result, keys must be associated with an Identity, creating an association with a meaningful origin. This is done with Certificates, which bind a key to an Identity. This binding is secured with a signature, and the private key used to sign the Certificate is denoted as the Signing Key or Certificate. The Signing Certificate is also signed by another private key, which is in turn managed by its own Certificate. This forms a Certificate Chain, and the chain continues until a Self-Signed Certificate is reached. This is the Root Certificate, also known as a Certificate Authority. It is signed using the private key that matches the contained public key, ending the chain. If an entity contains a copy of the Root Certificate and trusts its claim as an authority over its’ own identity, then all signatures in the chain can be trusted. This is how website identities are verified.
When a private key is compromised by an attacker, he/she would have the power to sign other Certificates and speak to their identity with the same trust granted to (and by) a Root Certificate. For this reason, the Public Key Infrastructure supports Certificate Revocation. Details are beyond the scope of this document, though it provides a way to replace compromised keys and retain the Certificate Chain’s integrity. The Internet depends on these relationships to form trusted connections, such as those you create when doing online banking.

**Secure Execution**
By chaining Certificates together using the described code signing process, Secure Boot is able to provide a way for application developers to offer firmware that OEMs can then securely execute. It does not require prior agreements or awareness of third parties – the inherent trust hierarchy enables this web of trust to manage such relationships.

**Loader Integrity**
Secure Boot firmware can only retain integrity if target hardware supports a certain minimal set of additional protection to ensure the boot loader is secure. This topic will be covered in more detail in later versions of this manual.

**Operation**
Secure Boot operates in one of three modes and uses a device key to manage application integrity. The system also integrates with hardware capabilities such as device ID, register locking, JTAG lockout, cryptographic acceleration, random number generation, write-only interfaces, and MMU/MPUs. At its core, however, Secure Boot operates in simple fashion.

**Operating Modes**

*Configuration Mode*
Secure Boot operating in *Configuration Mode* does not perform integrity checks on application and data before it is utilized. Secure Boot does however process service requests enabling you to provision the system in any one of several ways.

*Integrity Mode*
Secure Boot operating in *Integrity Mode* performs integrity checks on applications and data before it is utilized. Failures leads to Secure Boot Remediation execution.

*Error Modes*
On rare occasion, Secure Boot may enter an *Error Mode* of operation which requires human intervention to reset.

**Device Keys**
Secure Boot uses Device Keys to manage component integrity. As previously noted, integrity checking is managed with signing and the Chain of Trust realized by the Public Key Infrastructure. Device Keys should not be Root Certificates, though may be signed by a Root Certificate. This enables Secure Boot to offer key replacement services on the occasion that a private key is compromised. Without this, devices would likely need to be reprogrammed.
Device Key scope is, for this reason, rather significant. These and other similar considerations are covered in Appendix A – Advanced Considerations.

**Device Provisioning**

Before releasing your application on a new device, integrate the Secure Boot into your code using a valid RSA Public Key. This will utilize Self-Provisioning mode to validate the image on boot. Resources are managed with `fg_signelf.exe` (see below) and located here:

```c

\inc\fg_image_pubkey.h
  unsigned char fg_image_pubkey[] = {0x00, 0x00, 0x00, 0x00...};
  unsigned int fg_image_pubkey_len = 155;
\inc\fg_image_appsig.h:
  unsigned char fg_image_appsig[] = {0x00, 0x00, 0x00, 0x00...};
  unsigned int fg_image_appsig_len = 160;
```

With Self-Provisioning, Secure Boot recognizes valid entries for these items and automatically switches to **Integrity Mode**. When the RSA Public Key validates the firmware signature, the system will boot and run your application – else it enters a suspended (halted) state.

**Creating a sample project**

Your software package contains `fg_signelf.exe`, a utility application located in the `\bin` folder off the root of your distribution. We will use this along with Renesas e2 Studio to build a sample project to demonstrate Phase 1 of the multi-stage Secure Boot facility. Using this sample, the Renesas target board’s LEDs blink to reflect the state of the firmware’s integrity. The fully-integrated multi-stage bootloader will provide alternatives when integrity cannot be verified.

The following procedure uses an SK-S7G2 evaluation system. You should substitute your target hardware if it differs.

**Using fg_signelf for Self-Provisioning**

1. **Create a Test Application in e2 Studio**
   a. Start e2 Studio and create a new Synergy Project. File->New->Synergy Project
   b. Give a name to the project. In our example we call ours Proj
   c. Select your license key for e2-Studio, if you have been provided one
   d. In the Device Selection pane, choose S7G2 SK and keep the other default options
   e. On the Project Configuration pane, select the S7G2-SK blinky project template.
   f. Copy the provided fg_sboot_VSA folder’s contents into \workspace\Proj. Make sure that you have copied the file `\src\hal_entry.c` into \workspace\Proj\src replacing the version that was generated when you created the project.
   g. Choose Synergy Configuration pane’s BSP tab, then find the Properties tab in another pane, just under Synergy Configuration pane, and set Properties Heap Size to **0x2000**. NOTE: Depending on your environment, it may be necessary to utilize a larger heap. If your run into errors, increase the heap size and retry.
h. Click on **Generate Project Content** (on the top right hand corner of the Synergy Configuration pane)

i. Edit the project properties. Right click on **Proj** in the project navigation pane, select **Properties->C/C++ Build->Settings->Cross ARM C Compiler->Includes:** Add the following path to the list of include paths

   `<workspace>\inc`

j. While still editing the project properties, select **Cross ARM C Linker, Libraries:** Update the Library Search Path to and add the path

   `<workspace>\lib\Debug` **to the Library Search Path**

   Add `fg_port`, `fg_crypto`, `fg_cert`, and `fg_sboot` to **Libraries.**

k. Build your project to create `<workspace>\Proj\Debug\Proj.elf`

2. **Run fg_signelf to Configure Secure Boot**
   a. Open a command prompt and execute the following instruction:

      `<workspace>\Proj\bin\fg_signelf -genkeys -signapp -o:TestKey`

      When prompted to overwrite the existing key file, select y to overwrite the keys.

   b. Rebuild and run your e2 Studio firmware application

   The **first time you do this, the sample application will blink the LEDs at a high rate of speed, indicating that the firmware signature does not match the one generated** with `fg_signelf`.

   **Run fg_signelf again** with the same switches, and choose to overwrite the keys. When you subsequently execute the application, the LEDs will blink at 2Hz to indicate validated firmware.

   If you subsequently make changes to the application then rebuild and re-run, the LEDs will again blink at a high rate of speed. This is due to a change in the firmware. Because you didn’t change the signature, it no longer matches the flash program – and the demo reflects this with frequency.

   To sync the signature after making changes, first build your firmware, **run fg_signelf to generate the new signature**, and finally rebuild and run your e2 Studio project.

   Note that you will have to repeat this process if you make changes to the firmware, or if you change the keylength Secure Boot uses.

### Generating Keys

Generate an RSA key pair using `-genkeys` along with `-o:<filename>` to specify a target. To specify a keylength, use `-l:<length>`; acceptable values are 1024, 2048 (default), and 4096.

```
fg_signelf -genkeys -o:c:\keyfiles\test_priv -l:1024
fg_signelf -genkeys -o:z:\thumbkeyfiles\test_priv
```

The public key is stored in your e2 Studio project’s `\fg_sboot\keys` folder but the private key goes where you tell it to. Normally you would store private keys offline and keep them protected.
Overwriting Keys
When a key already exists in a target location, you are prompted to overwrite. Choosing ‘Y’ or ‘y’ moves the existing key to *.bak then places your new key in the proper target.

Keyfile Formats
Generated keys are put in a custom format that isn’t portable. Keys are generated using the Microsoft CNG default providers available with bcrypt.lib.

Import operations support DER-formatted keys and this same custom key format.

Signing Your .ELF File
Use -signapp to sign your application’s .ELF file. When not used with -genkeys, specify your private RSA key with -i:<filename>. As noted, this can be a DER formatted key or a key generated with this application, which uses a non-portable storage format.

The signing operation will find and use the 1\textsuperscript{st} occurrence of an .ELF file in your e2 Studio project’s Debug directory.

\textbf{NOTE:} You must run \texttt{fg_signelf.exe} from the proper location – your e2 Studio’s project bin folder.

Signing Operations
The signing process does several things at once, ultimately taking as input the proper keyfiles and offering as output all resources required for Secure Boot to Self-Provision.

First, the software finds your .ELF file in the e2 Studio Debug folder and parses this file to find the offset and size of the .text Section. This is the executable code the signing process attempts to validate – data sections (including .rodata) are not (yet) included.

The .text Section offset and size are used to read the .ELF image data and generate a signature that should match any similar operation computed against the same code programmed into target hardware flash memory. The signature is converted to a C character array and written to the appropriate header file so the e2 Studio project can be rebuilt, giving Secure Boot the resources required to perform boot image integrity validation.

The signing process also updates your target’s public key file – which is what Secure Boot uses to validate the signature. This takes from \texttt{\keys\fg_image.key_public} and in similar fashion creates a C character array that is put in \texttt{\inc\fg_image_pubkey.h}.

Hardware Integration

Additional Features

Secure Boot Interfaces
Appendix A – Advanced Considerations

Appendix B – Revisions

<table>
<thead>
<tr>
<th>Date</th>
<th>Author</th>
<th>Rev</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12/01/2015</td>
<td>J Poffenbarger</td>
<td>v1</td>
<td>1st e2_Studio release of Floodgate Secure Boot</td>
</tr>
<tr>
<td>12/13/2015</td>
<td>J Poffenbarger</td>
<td>v2</td>
<td>Phase-1 Secure Bootloader w/libs</td>
</tr>
</tbody>
</table>