View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Crossref

CHAPTER 9

Breaking the Boundaries with Dynamically Loaded Applications

Sometimes we stare so long at a door that is closing that we see too late the one that is open.

—Alexander Graham Bell

In previous chapters, we have studied the firmware architectures and security hardening features of the security and management engine. Let's recap the main design points:

- The security and management engine's firmware starts from boot ROM (read-only memory), which is not erasable and not modifiable.
- The boot ROM is the root of trust of the engine.
- The majority of the engine's firmware, including all applications, are stored in a flash device, together with other system firmware such as BIOS (basic input/output system).
- Firmware modules may be compressed with Huffman[1] or LZMA[2] to conserve the flash space. Firmware modules are not encrypted.
- Metadata of all firmware modules (including the kernel and various applications) is put together in a structure called *manifest*, also stored on the flash.
- The manifest contains SHA-256[3] digests for every firmware module. SHA-256 is one of the most frequently used *Secure Hash Algorithms*.
- The manifest is digitally signed by Intel with 2048-bit RSA[4] (Rivest, Shamir, and Adleman). The signature and the public key are both appended to the manifest.

During the boot process:

- The ROM verifies the RSA signature of the manifest. The SHA-256 fingerprint of Intel's public key is hard-coded in the ROM.
- The boot ROM verifies the SHA-256 digest of the first firmware module that is loaded from the flash.
- The integrity of subsequent modules is verified by one of the modules that have been verified and loaded previously in the boot process.
- When being loaded, a module performs the necessary initializations, and then creates a "worker" thread that waits for events. Most common events are system interrupts, HECI (host-embedded communication interface) messages initiated from the host, and service requests of other modules. Upon receiving an event, the module serves the event and waits for the next event.

By design, a module that runs on the engine must be compiled as part of the engine's firmware system, registered in the manifest, and preinstalled on the flash. The set of firmware applications and modules for a given product is determined at the time of compilation and cannot be changed after it leaves Intel's facility. From this perspective, the engine is a self-contained system, and doors are closed against loading new applications.

That being said, the engine is technically not a closed system, because it is capable of exchanging data with the external world at runtime. Notice that what is input to and output from the engine is only data, and may not be executable code. Running unauthorized code is a major violation of the security objectives of the engine.

Closed-Door Model

With the closed-door model, everything that can be executed on the engine is strictly controlled. Thanks to the integrity check mechanisms that are enforced during the boot process and runtime, the boundary of the engine is well guarded. It is very difficult for attackers to inject root kits and other malware to the system. The security architecture does not need to worry about possible vulnerabilities and potential flaws brought into the system by external applications. Therefore, the closed-door model is advantageous for security management.

Product quality-wise, the closed-door model makes validation simpler, because the functional testing is performed on predefined and constant configurations. Some of the common software and system problems, such as integration complexity and component compatibility, are not applicable.

Despite its security and stability, this design has its drawbacks:

- *Expansion of the engine's functionality is restricted by the flash space*. There are multiple products of the firmware, and their sizes vary between approximately 1.5MB and 5MB, which is fairly small considering the ever-growing number of features carried by the engine. Increasing the size limit is not free. In today's fierce competition environment, the BOM (bill of materials) cost is a pivotal consideration for all computer manufacturers. Raising the flash space consumed by the security and management engine requires flash chips of greater capacity, and hence adds the BOM costs for deploying manufacturers. When the size of the firmware binary reaches the maximum, new features can't be rolled out without taking current features out of the firmware.
- *Firmware update can be cumbersome*. Adding new applications to the engine or fixing bugs in existing modules requires more than Intel's development and validation effort. Rewriting firmware on the flash is a very privileged operation, and if done improperly, may render the system unbootable and result in a large number of support calls. Therefore, computer manufacturers have to test new firmware releases with all lines of products respectively and make sure there are no security or compatibility issues.
- *Intel is the sole development owner for the security and management engine*. Independent software vendors cannot build applications that run on the engine.

To address these drawbacks to some extent, newer versions of the security and management engine firmware include a module called the *Dynamic Application Loader*, or DAL for short. As indicated by the name, the DAL allows the engine to dynamically load and execute Java applets at runtime. The applets are not stored on the flash, but on the host's hard drive. With the DAL, the embedded engine is no longer a closed-door realm. The engine is now open to more flexibility and possibilities to be explored.

Meanwhile, more importantly, the security objectives of the engine remain the same and the security protection strength is not degraded because of the DAL.

DAL Overview

The DAL is implemented as an application in its isolated task in the firmware architecture. See Chapter 4 for details on the engine's task isolation design. Because the DAL loads an application from the host, it is active only when the host is awake. The DAL is not available if the host is in the sleep state. The relationships between the DAL task and other firmware components are depicted in Figure 9-1.

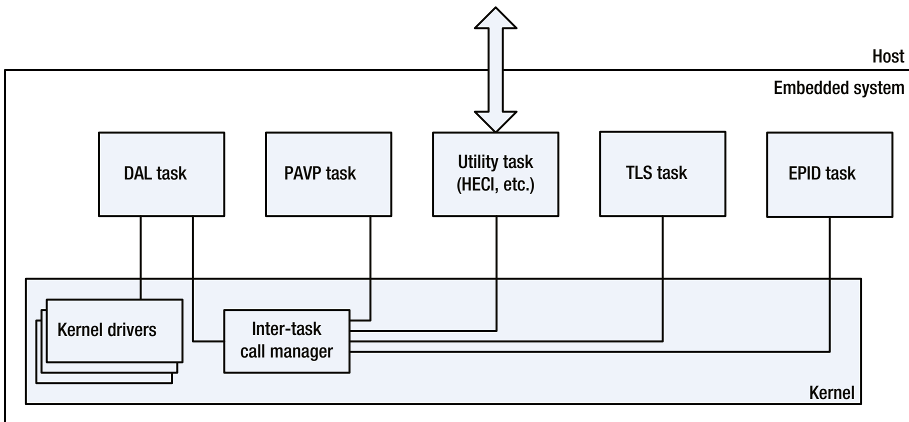

Figure 9-1. *The DAL task and its relationships with other firmware components*

To support functionality requirements of the loaded applets, the DAL task consumes several kernel services and other peer tasks:

- *Cryptography driver*: Provides implementations of popular cryptography algorithms, including AES[5] (advanced encryption standard), SHA, HMAC[6] (keyed-hash message authentication code), RSA, random number generator, and so forth.
- *Storage manager*: Secure nonvolatile storage for DAL management data and applet specific secrets.
- *Protected runtime clock*: Provides secure timer services for applets.
- *Image verifier*: The DAL replies on the kernel to verify the digital signatures on the dynamically loaded applications.
- *PAVP (protected audio and video path) task*: Some applets—for example, the applet that is part of the Intel IPT[7] (identity protection technology) solution—require secure display path that is not visible to software running on the host operating system. See Chapters 8 and 10 for more details on PAVP and IPT, respectively.
- *EPID (enhanced privacy identification) task*: Some applets realize functionalities that require Intel platform's hardware support. The EPID algorithm and SIGMA protocol are utilized to authenticate the platform and establish secure sessions between the host/ server application and the loaded applet. Refer to Chapter 5.
- *TLS (transport layer security) task*: Provides applets with secure PKI (public key infrastructure) support.

- *Utility task*: This task implements a number of interesting services, for example, CPU and chipset information, firmware status report, power states, HECI, and so forth. The HECI is the channel for the host to transmit the application's binary image to the DAL firmware task for execution.

Due to these dependencies, the firmware product that features the DAL must also support these tasks. The DAL task is not consumed by any other firmware modules. Note that the DMA (direct memory access) is not used for transmitting applets from the host to the engine. To further minimize security risks, the DMA driver is not available to applets to invoke.

DAL Architecture

The DAL is essentially a Java virtual machine that enables the operation of Java applets in the security and management engine's firmware environment. The Java applets in bytecode implement their designed functionalities that can be executed in the firmware. The components that make up the DAL feature are shown in Figure 9-2.

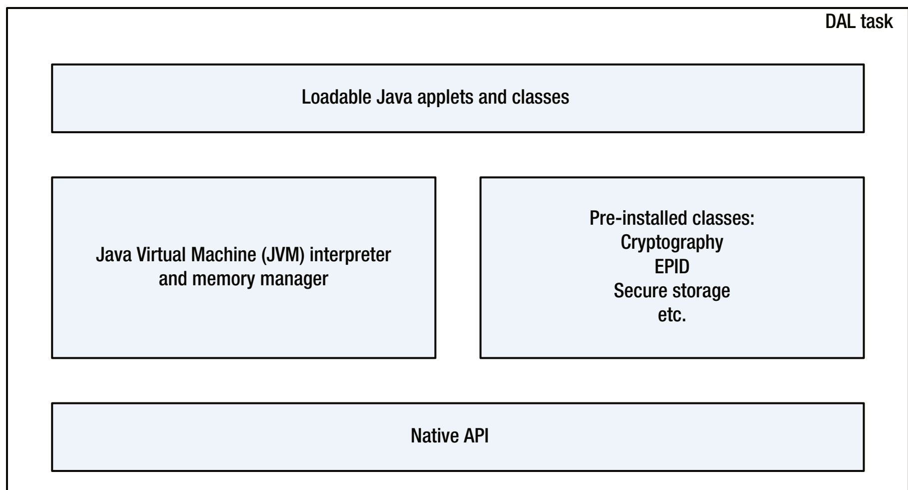

Figure 9-2. *Components of the DAL*

A service layer (classes) can also be loaded from the host together with the applets. The service layer may realize utilities such as HECI. In addition, the DAL task preinstalls select services, such as cryptography and EPID, without needing to load from the host. Those services are specific to the firmware engine and they are expected to be used by most applets.

The native API (application programming interface) component receives the Java API calls, performs conversion, and in turn calls appropriate kernel API or other tasks. It serves as a proxy so the Java classes do not need to be aware of the engine's specific interfaces.

Loaded Java applets are free to take advantage of various services offered by the engine, but there is no guarantee regarding performance, due to a few facts. Firstly, to save resources, when multiple applets are loaded to the system, the DAL and the installer application may decide to temporarily unload an applet and reinstall it later as needed. This procedure may delay the applet's responses to requests from the host. Secondly, the DAL task shares hardware resources with other firmware capabilities running in the security and management engine. The engine is a multithreaded environment, and the amount of clock cycles allocated to a specific thread is not guaranteed.

With these considerations, the DAL is not intended for loading major features to the engine. Rather, it is designed for offloading critical security components of a consumer solution, for example, the Intel IPT. In contrast, loading the entire or a large part of the AMT (advanced management technology) firmware application from the host at runtime is not an appropriate usage of the DAL.

Loading an Applet

A Java applet package can be obtained from various resources, such as software vendors' distributions and web sites. On Windows, a host software program loads applets to the security and management engine through the *Intel dynamic application loader host interface service*. Because the engine does not persistently store applets in its nonvolatile memory, an applet must be reloaded when the host power cycle is reset. However, it is worth emphasizing that the DAL firmware treats the first time that an applet is loaded to the engine differently from consequent loads of the same applet in the future.

In the engine's secure nonvolatile storage, the DAL maintains a database of all applets that it has loaded at least once and their metadata. An entry of the database records, among other attributes, the unique identification of the applet, its version number, and its security version number. When an applet is loaded for the first time, a new entry is created in the database for the new applet. The entry is examined and updated as necessary.

Upon receiving a request from the host to load an applet, the DAL first checks whether an instance of the applet with the same identification has been loaded previously in this power cycle. If so, the host must first ask the DAL to unload the applet before loading it to the DAL again. The DAL does not voluntarily unload an applet unless the host requests so. The reason for reloading an applet may be to update the applet to a newer version. This is shown in Figure 9-3.

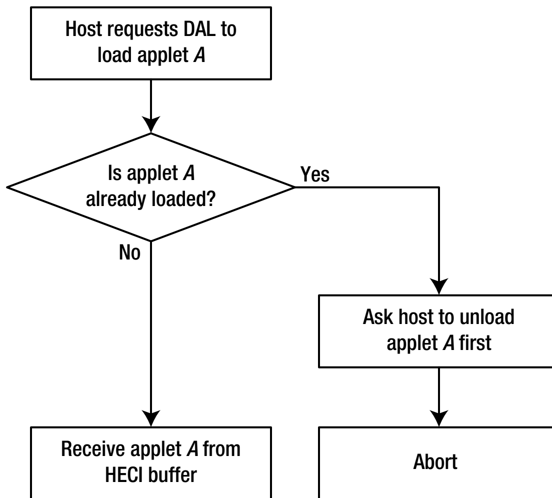

Figure 9-3. *Handling an applet load request*

A loadable Java applet is always packaged with its corresponding manifest. The structure of the manifest is similar to what is shown in Figure 4-1 in Chapter 4. Specifically for security, the following fields are critical in the loading process:

- Applet identification.
- DAL flag, indicating this is a DAL manifest.
- Version number.
- Security version number. A security version is assigned to every applet release. If vulnerabilities are found in an applet, then the new applet release that fixes the vulnerabilities will be assigned an incremented security version.
- RSA signature of the manifest.
- RSA public key.
- SHA-256 digest of the applet.

The applet to be loaded also specifies the minimum version of the engine's firmware that is required to run this applet. Earlier firmware releases may not be equipped with the necessary infrastructures to support the applet. The process of loading an applet is illustrated in Figure 9-4.

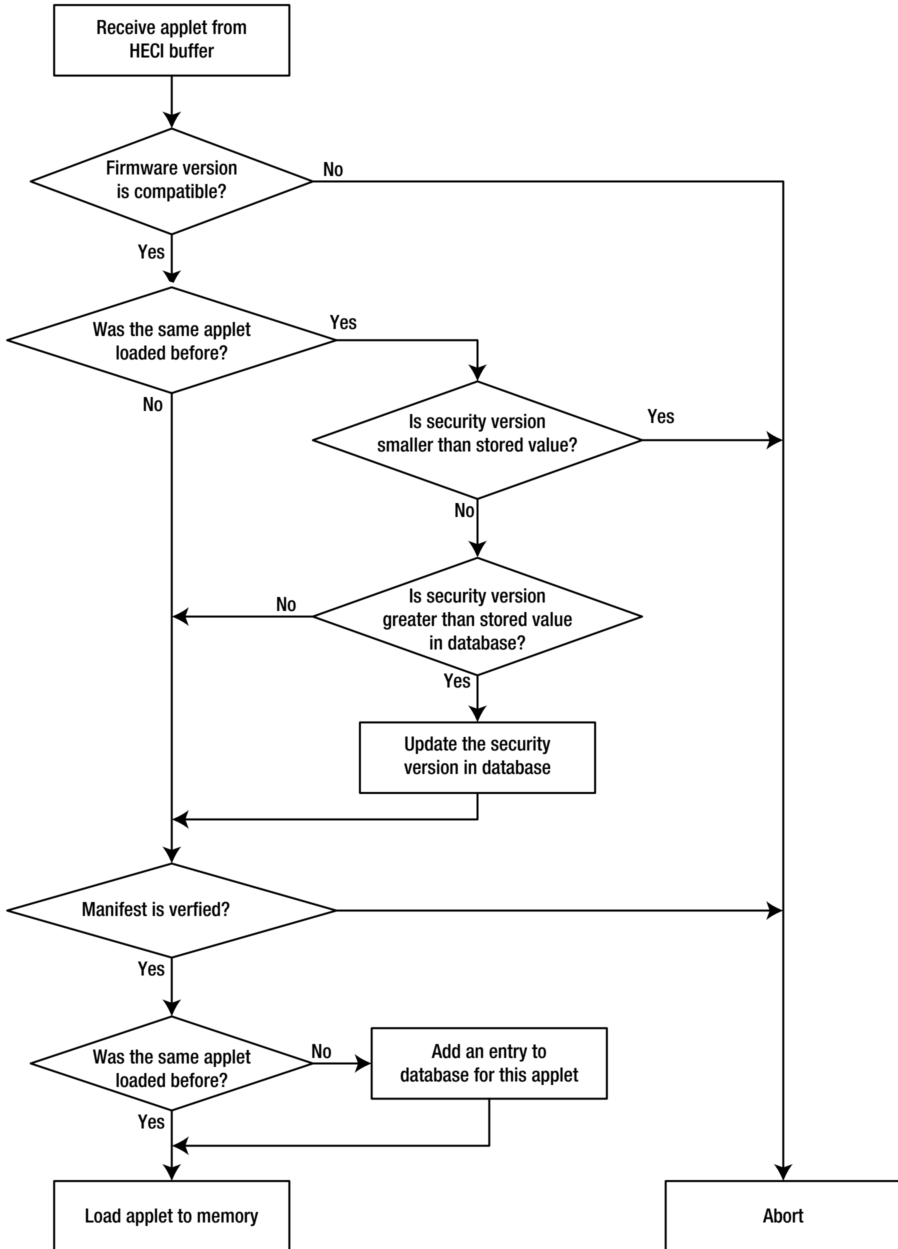

Figure 9-4. *Process of loading an applet*

The loading process starts from the DAL task receiving the complete applet package, including the manifest, from the host in the HECI buffer. As introduced in Chapter 3, a HECI message has limited capacity, depending on the engine's configuration. If the size of the applet package is greater than the capacity of a HECI message, then the package will be split and come in multiple messages.

After possessing the applet, the DAL first verifies that the firmware currently operating on the engine is capable of executing the applet. The DAL aborts the loading if the firmware is too old to support the applet. A firmware update will not be automatically launched in this case. The user must manually update the engine's firmware in order to run the applet.

If the applet has been loaded before, then the DAL makes sure that its security version is not smaller than the one stored in the database. If this is not the case, then it may be a rollback attack that exploits the vulnerability in an older applet release and the DAL shall reject to load the applet. If the security version is greater than what is shown in the database, then the DAL updates the database with the newer value. If the DAL has never seen this applet before, then it creates an entry for it in the database after the integrity check passes.

The manifest validation is performed by invoking the kernel API. The manifest must be signed with the same RSA key that signs the manifest for the engine's firmware image loaded from the flash.

Secure Timer

The DAL provides applets with secure timer services that measure the time elapsed between a *Set timer* call and a *Get current timer value* call. When multiple applets are running simultaneously, each applet may create one or more independent timers. The timer is useful for applications that must enforce durations—for example, a one-time password that expires every 30 seconds.

Host Storage Protection

The engine is allocated with only limited flash space for its data partition. Therefore, to reduce the flash footprint, it is recommended that the Java applets do not store data on the flash. Instead, applets' nonvolatile data, especially if its size is large, should be placed on the host's hard drive.

To facilitate and protect the host storage mechanism, the DAL provides an encryption key and an integrity key for every applet. A typical usage would be to encrypt data using the encryption key, append an HMAC-SHA-256 signature (generated using the integrity key) to the encrypted data, and then send the blob of encrypted data and the signature to the host for storage. To retrieve the data, the applet simply fetches the blob from the host, verifies the HMAC signature using the integrity key, and then decrypts using the encryption key. Optionally, anti-replay protection can also be applied to data blobs if necessary, to mitigate rollback attacks (replacing a blob with an older version).

The encryption key and the integrity key are persistent for the same applet even if the engine has gone through power cycles. Derived from a bit string that is randomly generated when the DAL is initialized for the first time on a platform and the applet's unique identification, the keys are unique for the applet that runs on the specific

platform. In other words, an applet is not able to make use of a data blob that was created by another applet; cloning a data blob from one platform to another would not pass the integrity check. It is up to individual applets to decide the proper reaction to take upon blob failures.

Security Considerations

Naturally, alongside the openness of the DAL come new security concerns. Specific security requirements are set for safeguarding the engine with the existence of the DAL:

- Applets can be executed only after being loaded by the DAL firmware application. Modules in a manifest that is intended for the DAL shall not be executed directly on the engine's embedded processor.
- The DAL shall not load manifests that are intended to be loaded by the engine's regular boot process.
- The DAL shall enforce context separation among distinct applets.
- The DAL shall record the greatest security version numbers for each applet respectively, for rollback attack detection.
- An applet shall follow security design guidelines for regular firmware applications, such as using minimum privileges, minimizing attack interfaces, and so on.

The first two bullets are the most critical requirements. Because the applets' manifests are signed with the same RSA key that signs the firmware image, the architecture must mitigate image replacement attacks where an attacker replaces the firmware image on the flash device with an applet image, which will pass the signature verification conducted by the boot ROM.

The countermeasure employed by the architecture is to introduce a "DAL" flag in the manifest. The firmware's boot process will not load a manifest with a DAL flag set. Conversely, the DAL will not load a manifest if its DAL flag is not set.

Reviewing and Signing Process

Applets may be developed by Intel or third-party software vendors. The process for reviewing and signing an applet is the same regardless of whether Intel or a third-party is the applet developer. The high-level process is described in Table 9-1.

***Table 9-1.** Applet Reviewing and Signing Process*

Stage	Name	Activity
1	Applet creation	Vendor creates the applet.
2	Applet review	Intel reviews the applet for functionality, security, and privacy.
3	Manifest creation	Intel creates preproduction manifest and provides to the vendor.
4	Preproduction testing	Vendor tests and debugs the applet on a preproduction security and management engine. Sometimes a simulator is used instead. Go back to stage 1 if any change is made to the applet.
5	Presigning	Intel makes sure the content of the applet is identical to what is in the final preproduction manifest.
6	Signing approval	Approvers review and sign the manifest. Critical manifest parameters (such as security version number and DAL flag value) are displayed to approvers for a final review.
7	Signing	The signing tool replaces the RSA public key and the signature in the preproduction manifest with a production RSA public key and signature.
8	Production testing	Vendor tests the applet on a production security and management engine.
9	Ready for distribution	Vendor is ready to distribute the applet.

References

1. D.A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the I.R.E.*, September 1952, pp. 1098–1102.
2. Igor Pavlov, "LZMA Software Development Kit," http://7-zip.org/sdk.html, accessed on December 12, 2013.
3. National Institute of Standards and Technology, "Secure Hash Standard (SHS)," http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf, accessed on November 17, 2013.
4. RSA Laboratories, PKCS #1 v2.1: RSA Cryptography Standard, ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf, accessed on November 17, 2013.

5. National Institute of Standards and Technology, "Advanced Encryption Standard (AES)," http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, accessed on November 17, 2013.

6. National Institute of Standards and Technology, "The Keyed-Hash Message Authentication Code (HMAC)," http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf, accessed on November 17, 2013.

7. Intel Identity Protection Technology, http://ipt.intel.com, accessed on April 20, 2014.