Verifying Type and Configuration of an IBM 4758 Device

A White Paper

S.W. Smith
Secure Systems and Smart Cards
IBM T.J. Watson Research Center

February 8, 2000

1. Introduction

The IBM 4758 is a "programmable secure coprocessor." This term embodies two somewhat orthogonal concepts:

- As a *secure coprocessor,* the device provides a haven for security and cryptographic operations, safe against some specified level of physical attack.
- As the device is *programmable,* what these operations are (the *software configuration*) can be established in a number of ways, to accommodate various customer scenarios.

If a customer installation depends on the security properties of a particular type of 4758 device with a particular software configuration, it is critical that they verify that the black box inside their machine really is that type of device, configured in that way.

The purpose of this white paper is to quickly summarize how to do this verification.

- **What It Is.** First, this paper presents the parameters describing *what* the device is. Section 2 presents the device types for the 4758 family; Section 3 presents the principal parameters describing software configuration.
- **How to Tell.** Then, Section 4 presents the mechanisms for authenticated access to this data.

However, the question of when and where to perform this verification depends on the security architecture and threat model of the actual deployed system. Section 5 presents some of these consideration.

Note: This white paper reflects the fact that, as of this writing, the Model 2 family of devices are only available as prototypes (under special arrangements). When non-prototype devices become available, this white paper will be duly revised.

2. Hardware

There are five basic production variations of the 4758:

- Model 1: the initial 1997 release, with physical security validated at FIPS 140-1 Level 4
- Model 13: the same basic coprocessor as Model 1, but with physical security validated at FIPS 140-1 Level 3

- Model 1 8V1152: the same basic coprocessor as Model 1, but with physical security weaker than Model 13 (released under special arrangement to an OEM customer)
- Model 2: our follow-on coprocessor (with faster crypto and CPU than Model 1), with physical security intended to be validated at FIPS 140-1 Level 4
- Model 23: the same basic coprocessor as Model 2, but with physical security intended to be validated at FIPS 140-1 Level 3

Model 2 and Model 23 devices will be available in both 5-volt and 3.3-volt versions.

Note that, as of this writing, Model 2 and Model 23 devices are only available as prototypes, under special arrangement.

The precise type of a particular 4758 device is indicated in its *vital product data (VPD)* data structure. Table 1 summarizes this correspondence. (Section 4 will present how to securely obtain this information.)

3. Software

The 4758 architecture partitions the in-card software into four layers, Layer 0 through Layer 3.

Layer 0 and Layer 1 contain *Miniboot*, the IBM-owned security bootstrap and configuration code. This code runs when the card is booted, then goes away until the next device boot. The roles performed by these layers are the same in each released device (although the specific versions of the code may vary). Layer 0 resides in ROM and cannot be changed; Layer 1 resides in protected FLASH and can be updated via a signed command from IBM.

Layer 2 and Layer 3 contain the software that controls what the card does after bootstrap. (Layer 2 is generally intended for supervisor-level operating system/system software code; Layer 3 is generally intended for user-level application code.) In contrast to Miniboot, both the role *and* the owners of this code can change from card to card.

Note that our security architecture does not *depend* on supervisor-user separation between Layer 2 and Layer 3. Rather than permitting multiple applications which might attack each other, we allow only one—which therefore, should it find a way to breach the separation, can only attack itself. (Ongoing FIPS validation work examines our Layer 2 software only in the context of a specific application program). It is our Miniboot security configuration software—which has withstood FIPS 140-1 Level 4 formal scrutiny—that decides who controls the other segments and what code should be loaded, and which authenticates the configuration of the card. No code that runs after Miniboot finishes can modify Miniboot or access its private keys; *hardware* provides this protection: before completing its work, Miniboot advances the hardware "ratchet" lock, which denies all access until the next device reboot causes Miniboot to run again. (Our security architecture paper[1] provides more details.)

Hence, the question of "what is this card's software configuration" reduces to two pairs of questions:

- Who is the owner of Layer 2? Of Layer 3? (See Section 3.1 below.)
- What code has been installed in Layer 2 by its owner? In Layer 3 by its owner? (See Section 3.2 below.)

As Section 3.3 discusses, *it is critical that one address both issues.*

3.1. Layer Owners

To answer the first question, Layer 2 and Layer 3 each have a two-byte *ownerID* field. When the devices leave the factory, these fields indicate that neither layer is owned.

If an entity (either within IBM or external) wishes to issue Layer 2 code for 4758 devices, they generate a public-key/private-key keypair, and (upon suitable arrangement) receive the following from IBM.

[1] Smith, Weingart. "Building a High-Performance, Programmable Secure Coprocessor." *Computer Networks* (Special Issue on Computer Network Security). 31: 831-860. April 1999.

- a Layer 2 ownerID (unique among all Layer 2 owners), assigned by IBM;
- the appropriate Miniboot command (signed by IBM) to grant ownership of an unowned Layer 2 to that entity;
- the appropriate certificate (signed by IBM) to enable that entity to issue their own Miniboot commands to change Layer 2, using their own keypair. However, these Miniboot commands are accepted only by devices whose Layer 2 is owned by that entity.

If an entity (either within IBM or external) wishes to issue Layer 3 code for 4758 devices, they first need to pick a *parent*: a Layer 2 owner whose code they will build on. They then generate a public-key/private-key keypair, and (upon suitable arrangement) receive the following from the *Layer 2 parent* they have chosen:

- a Layer 3 ownerID (unique among all Layer 3 owners depending on that Layer 2 parent), assigned by the Layer 2 parent;
- the appropriate Miniboot command (signed by that Layer 2 parent) to grant ownership of an unowned Layer 3 (in a card whose Layer 2 is owned by that parent) to that entity;
- the appropriate certificate (signed by that Layer 2 parent) to enable that entity to issue their own Miniboot commands to change Layer 3, using their own keypair. However, these Miniboot commands are accepted only by devices whose Layer 3 is owned by that entity. and whose Layer 2 is owned by their parent.

Table 2 summarizes some principal ownerID values currently assigned. (Again, Section 4 will present how to securely obtain this information.)

3.2. Layer Contents

To answer the second question (about what code has been installed), Layer 2 and Layer 3 each have a number of fields describing their contents:

- an 80-byte image name field
- a 16-bit image revision field
- a 20-byte image hash field.

The image name and image revision fields are chosen by the Layer owner. In contrast, the image hash field is the exact hash of the image itself.

Table 3 summarizes the principal Layer 2 image names. Table 4 summarizes the principal Layer 3 image names. (Again, Section 4 will present how to securely obtain this information.)

3.3. Importance of Verifying Both Owner and Contents

To describe the configuration of a typical desktop computer, one usually just lists the names of the installed programs. However, this intuitive procedure is insufficient for a programmable secure coprocessor like the IBM 4758; for secure operation, it is important to verify both the ownerID as well as the image name. This is for several reasons.

- Different owners may have different image replacement policies.

  For example, standard IBM CP/Q++ has been issued both under our production Layer 2 ownerID as well as our developer's toolkit ID. However, the developer's toolkit ID also permits state-preserving swapping between the standard CP/Q++ and one equipped with a kernel-level debugger. This is desirable behavior for a development environment but dangerous for many production environments; the Layer 2 ownerID distinguishes between the two.

- In theory, different owners may use the same name for different images.

  Recall that the Layer N owner *chooses* the image name for Layer N—and assigns the ownerIDs for its Layer $N + 1$ owners.

4. How to Tell

Section 4.1 and Section 4.2 present the basic mechanisms by which Miniboot communicates authenticated configuration information to outside entities. Section 4.3 presents how the standard host-side utility uses these mechanisms. Section 4.4 presents some new ways to access this data in Model 2 devices.

4.1. Certification of Device Keys

Within each untampered 4758 device, the Miniboot 1 security software maintains a certified keypair. Tamper events (or failures that render Miniboot 1 unrunnable) cause the private key (of this keypair) to be zeroized. Thus, if a message is signed with the private key matching a properly certified public key, then one can be assured that this message came from Miniboot 1 within an untampered 4758 device.

The certificate chain that establishes the authenticity of this keypair also provides further information on the type of device and the release history of Miniboot 1 within that device. In particular, IBM maintains an overall *4758 root* keypair, and, for each class of 4758 (e.g., each entry in Table 1), a *class root* keypair. (The root public key will be available on the 4758 web site.)

At the final stage of manufacture, each IBM 4758 device is *initialized*:

- Miniboot 1 within the device generates a public-private keypair, and retains the private key within tamper-protected memory.
- IBM certifies the public key with the class root keypair for that device class.

The certificate that IBM creates for the device includes the device type; however, we use a separate class root for each class in order to provide a second barrier to a customer accidentally mistaking an untampered device of one class for one of another.

When Miniboot 1 in a device atomically updates itself (in response to a correctly signed update command), it generates a new keypair for its successor, and certifies the new public key with its current private key. (This provides some protections should a defective Miniboot 1 be accidentally issued.)

Thus, within each untampered device, Miniboot 1 has a keypair, and chain of one or more certificates supporting that keypair, going back to the IBM class root, and then to the IBM root.

4.2. The Basic Mechanisms

Miniboot provides some basic mechanisms to provide secure statements of the identity and configuration of a 4758 device.

Each time the 4758 boots, host-side software can issue zero or more queries to Miniboot 1. (Note that 4758 device goes through bootstrap every time it comes up, and also upon explicit host request.)

One of these queries is the *Certlist Query*, which will return the certificate chain supporting the current Miniboot keypair in that device. (This certificate chain also indicates what type of 4758 device this is—Table 1.)

Another one of these queries is the *Signed Health Query.* As part of this query, the caller provides a nonce of its own choosing.

Miniboot 1 in the device will respond with a signed data structure, which includes:

- the nonce provided by the caller, so the caller can be assured the response is fresh;
- the VPD, so the caller can determine device type (Table 1)
- whether Layer 2 and Layer 3 are owned and have reliable contents
- the Layer 2 OwnerID and Layer 3 OwnerID, so the caller can determine who controls the software configuration (Table 2)
- if Layer 2 is owned and reliable, the name of the image it contains (Table 3)
- if Layer 3 is owned and reliable, the name of the image it contains (Table 4)

By verifying this signature against the public key for Miniboot in that device, and verifying the certificate chain taking that public key back to the IBM root, the caller can be assured that this device really is an untampered member of the 4758 family—and that the information in the data structure can be believed.

4.3. Using CLU

The *validate* command of the IBM-supplied *Coprocessor Load Utility (CLU)* uses these Miniboot query mechanisms to verify and display device identity and configuration.

Users need to specify a *validation file*, which contains a certificate for the class root public key for that device class. (The validation files for standard production classes are shipped along with the CLU program.) CLU will validate this certificate against the overall IBM 4758 root public key, which is hard-coded into CLU. Sufficiently paranoid users should take steps to ensure the integrity of the CLU executable.

The new CSU CAV tool performs a similar function.

4.4. Using Outbound Authentication

The Miniboot queries of Section 4.2 provide secure statements about device type and configuration only during device bootstrap. (Again, note that 4758 device goes through bootstrap every time it comes up, and also upon explicit host request.) However, scenarios exist where it would be useful for the Layer 3 application *while running* to be able to securely establish, to an external entity, the identity of that application, and the type and configuration of its underlying platform.

To address this need, Miniboot 1 and CP/Q++ for the Model 2 family of devices will provide an *outbound authentication* API to Layer 3, by which Layer 3 can have keypairs generated for its on-device use. These keypairs will be certified by a trust chain that goes down to Miniboot 1 on that device, and then back to the IBM class root, then back to the IBM Root. This trust chain will fully specify the relevant parameters—VPD, layer ownerIDs, image names—in an authenticated way.

Many classic security problems—including the authenticating device configuration—are characterized by the challenge: how to achieve authenticated communication with an entity in an environment with various hostile trust properties. Numerous cryptographic protocols exist that solve these problems—once the entity has a keypair certified to its identity. By giving an on-card application access to keypairs bound to that on-card application, in a particular software configuration on a particular device, this API enables the software designer to make full use of these protocols.

- For one example, an application could, when it is initialized, have a query-signing keypair created. Then, at any point during execution, the application could itself sign health queries about its status, and return this signed data along with the certificate chain.

  This frees the designer from worrying about attacks between device boot and application execution.

- For another example, perhaps the device's host H_U is untrusted, but another remote machine H_T is trusted. The application could have a signing keypair created, and then use this signing keypair to establish a shared

symmetric key with code at H_T. With appropriate design, H_T and the on-card application could then use this key to establish a secure communication channel—protecting this interaction from software attacks on H_U, and attacks on the network between H_U and H_T.

In particular, this frees the designer from worrying about attacks between verifications.

- An application designer could even extend the above example to establish secure sessions between a program on one device, and a program on another device—despite potential compromise of the intervening channel. (The OA API even provides an on-card program with its device class public key—so it can authenticate peers.)

(More information on this new API will be available in forthcoming software manuals from IBM.)

5. When and Where to Verify

The IBM 4758 was intended to be a platform upon which software designers could build secure coprocessing and cryptographic applications. The verification techniques discussed in this white paper are hooks which the designer can use—along with other techniques in his or her arsenal, such as host-side integrity checking, intrusion detection, independent audit, and procedural controls—to secure his or her application. However, how a designer should use these hooks depends on the overall security architecture. What threats does this design need to address? Which elements—machines, code, networks, personnel—of the system are trusted?

For example, a designer might want to consider questions such as the following:

- Can an adversary substitute a fake card in the middle of the night?
- Can an adversary modify the host software, so that it directs some or all host-card communications to some entity other than the genuine card still in the system?
- Can an adversary modify the software that verifies the device configuration?
- Can an adversary take control of the host machine after the host machine has booted?
- Can an adversary attack the network between a trusted host and the host with the device?

If two different designers answer these questions differently, then the appropriate security countermeasures will also differ.

To be most effective, this security tool—like any other security tool—should be used only in the context of a well-thought-out overall strategy, directed against a well-defined set of threats.

Acknowledgments

The author gratefully acknowledges helpful advice from Todd Arnold, George Dolan, Joan Dyer, Jonathan Edwards, Dave Evans, Richard Moore, Elaine Palmer, and Ron Perez.

Family	*Model*	*FIPS 140-1 Physical Security*		*VPD "Description field"*
Model 1 family	Model 1	Level 4		"IBM 4758-001 PCI Cryptographic Coprocessor"
	Model 13	Level 3		"IBM Crypto Coprocessor FIPS140-1 LVL 3 HDW"
	8V1152	N/A		"IBM 4758-001 RPQ8V1152,NO TAMPER DETECTION"
Model 2 family prototypes	Model 2	Level 4 (intended)	5V	"IBM4758-002 5V Prototype No Tamper Detect"
			3.3V	"IBM4758-002 3.3V Prototype No Tamper Detect"
	Model 23	Level 3 (intended)	5V	"IBM4758-023 5V Prototype Not Secure ROM"
			3.3V	"IBM4758-023 3.3V Prototype Not Secure ROM"

Table 1 Device descriptions for the 4758 production family. Note that, at this writing, Model 2-family devices are only available as prototypes under special arrangement. Some early Model 13 devices misspell "Crypto" in their description fields.

Layer 2 OwnerID	*Layer 3 OwnerID*	*What this Means*
00: UNOWNED	(any value)	Layers 2 and 3 are unowned
(any value)	00: UNOWNED	Layer 3 is unowned
02: IBM CCA Standard Product	02: IBM CCA Standard Product	Standard CCA configuration
	(various UDX, OEM values)	Customized CCA configuration
03: Development	06: Development	Standard Developer's Toolkit configuration
01: IBM Research	(any value)	Research OS
22: IBM Research	(any value)	Research OS
243: IBM Custom Products	14: PKCS-11	PKCS-11 configuration
	(other values)	other IBM/OEM custom configurations

Table 2 Principal OwnerIDs for the 4758 Family. Note that *all values are decimal!*

Layer 2 OwnerID	*Layer 2 Image Name*	*What it Means*
02: IBM CCA	"CCA <release> SEGMENT-2 <date, flags>"	Standard IBM CP/Q++ OS/System Software
03: Development	"CCA <release> SEGMENT-2 <date, flags>"	Standard IBM CP/Q++ OS/System Software
	"CP/Q++ <release> with Probe <date, flags>"	Special IBM CP/Q++ with Debugger Probe
02: IBM Custom	"CCA <release> SEGMENT-2 <date, flags>"	Standard IBM CP/Q++ OS/System Software

Table 3 Principal Layer 2 image names

Layer 2 OwnerID	*Layer 3 OwnerID*	*Layer 3 Image Name*	*What it Means*
02: IBM CCA	02: IBM CCA	"CCA <release> SEGMENT-3 <date, flags>"	Standard IBM CCA Application
03: Development	06: Development	"<filename> <len> <timestamp> (Druid <version>)"	Toolkit-loaded Development Code

Table 4 Principal Layer 3 image names