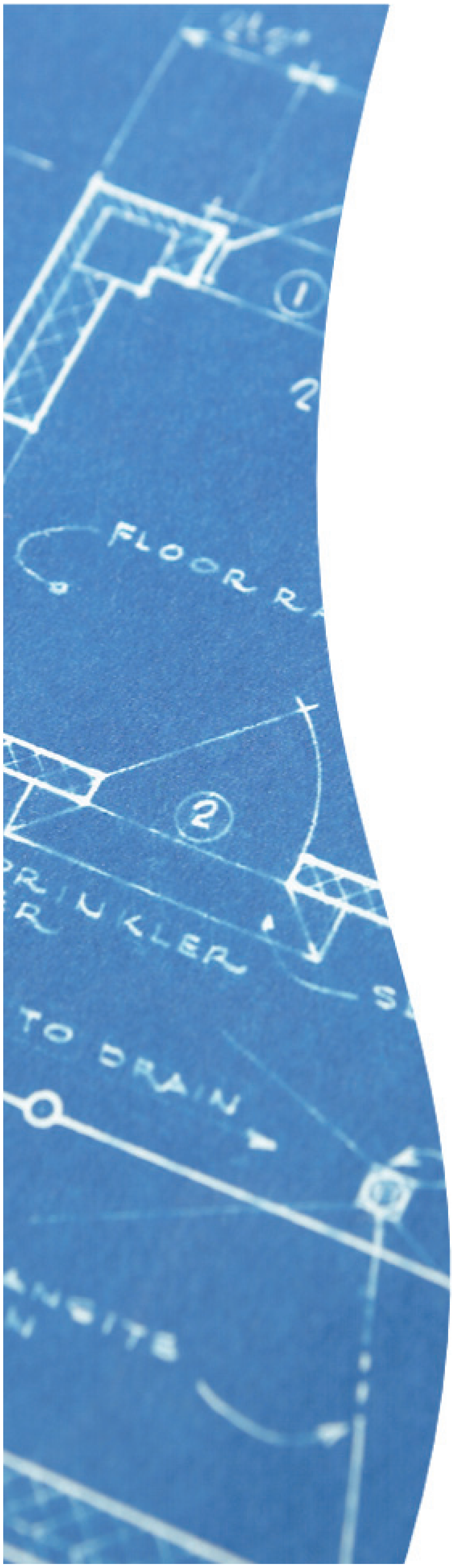

CRYPTOGRAPHIC SOLUTIONS FOR FINANCIAL SERVICES

Using the Sun™ Crypto Accelerator 6000 Card

Serge Nadon and Joel Weise
Sun Microsystems

Sun BluePrints™ Online — February 2008

Part No 820-4019-10
Revision 1.0, 1/31/08
Edition: February 2008

Table of Contents

Chapter 1
Introduction

The Sun™ Crypto Accelerator 6000 PCI-E card (SCA 6000 card) is a combined cryptographic accelerator and Hardware Security Module (HSM) that can be used to accelerate Secure Sockets Layer (SSL) and IPSec sessions, as well as perform various financial services related cryptographic functions. Qualified as a FIPS 140-2 level 3 device, the SCA 6000 card is designed to prevent the disclosure or corruption of cryptographic keying material, intermediate cryptographic results, or other sensitive data. A direct key loading interface is incorporated to enable the secure entry of keying material. Since sensitive keying material does not cross system, network, or application boundaries, potential avenues of interception and attack are eliminated.

The security of a cryptographic device is dependent upon not only the anti-tamper circuitry and design of the device itself, but also the processes and procedures used to initialize the device, and perform key management and application level transactions. This Sun BluePrints™ article assumes a working knowledge of financial services and contemporary security issues, and discusses some control mechanisms. It describes some of the processes and procedures needed to make the SCA 6000 card available to an application performing financial services transactions such as PIN management and verification, and card verification. In particular, the following topics are discussed:

- Bootstrapping the card, including initializing the card master key and key store
- Generating a financial services master key and key exchange key
- Performing financial services related key management activities
- Performing various application level financial services commands

Several examples are presented to illustrate the practical use of the SCA 6000 card:

- Master File Key (MFK) component installation
- Key Exchange Key (KEK) component installation and use
- Zone Working Key (ZWK) import
- PIN Verification Key import
- PIN Verification Key use

The cryptographic key hierarchy used by the SCA 6000 card and relevant applications is also described, along with the methods used to retrieve key objects. Note that the SCA 6000 card does not manage or store individual cryptographic keys. Only the MFK is stored within the SCA 6000 card—other keys must be managed as data objects by an appropriate user application.

System Configuration

Table 1-1 describes the system configuration used to illustrate the processes in this document.

Table 1-1. System configuration

User-Level Providers
• /usr/lib/security/$ISA_pkcs11_kernel.so • /usr/lib/security/$ISA_pkcs11_softtoken.so
Kernel Software Providers
• des • aes • arcfour • blowfish • sha1 • md5 • rsa • swrand • sha2
Kernel-Level Crypto Provider
• mca/0

Chapter 2
Design Principles

The SCA 6000 card is designed to ensure compliance with relevant ANSI, ISO, and other standards governing financial services related cryptographic processing, such as PIN processing or credit card verification. The SCA 6000 board design is based on the following principles.

- *Key separation and compartmentalization of risk*
  Keys are used only for specifically defined functions, limiting the damage that can result from a compromised key. To meet this requirement, functional key type information is associated with each financial services key. This information is used by the SCA 6000 card to enforce key separation and prevent one type of key from being used for a different purpose. The generation and importing of the keys defined in the key hierarchy are permitted, with the SCA 6000 card ensuring they are used for specific operations only.

- *Key forms*
  Clear text (human readable) keys stay within the SCA 6000 card, with the exception of those existing as at least two separate components, each under the control of a different security officer. When not stored in the card, or when in component form, all keys are enciphered with a key of equal or greater cryptographic strength.

- *Direct key loading*
  For security, the MFK and KEKs can be entered directly into the SCA 6000 card with a direct input device connected to a serial port on the board. For example, the Termiflex OT/30 handheld device is certified for use with the SCA 6000 card. The MFK and KEKs are entered in component form by unique security officers, enabling the SCA 6000 card to meet key management requirements relating to split knowledge, dual control, and data corruption.

  Split knowledge means that cryptographic keys in multiple component form are entrusted to, and controlled by, more than one key custodian—and that one custodian can never learn or know a key component managed by another custodian. *Dual control* ensures that no single person has the capability to obtain, determine use, alter, or ascertain a cleartext key, or more than one cleartext key component. Because key components never transit or are processed by any system, network, or application when input directly to the SCA 6000 card, they cannot be intercepted, modified, destroyed, or disclosed.

Key Hierarchy

The following types of financial services keys are supported:

- *Master File Key (MFK)*
  The MFK encrypts other operational keys when they leave the HSM, such as those used for storage. MFKs are entered into the SCA 6000 card in component form with a direct input device. The MFK never leaves the secure confines of the HSM.
- *Key Encryption Key (KEK)*
  The KEK encrypts other keys for key exchange operations, such as key or key material import and export. The KEKs are entered into the SCA 6000 card in component form with a direct input device.
- *PIN Encryption Key (PEK)*
  The PEKs encrypt PINs during interchange processing. Two types of PEKs are supported.
  - Terminal PIN Keys (TPKs) encrypt PINs on the terminal side of the transactions during acquisition. Examples of terminals include ATM and POS devices.
  - Zone Working Keys (ZWKs) encrypt PINs during the interchange between different financial institutions.
- *PIN Verification Key (PVK)*
  The PVKs verify PIN values, such as PIN Verification Values (PVVs).
- *Card Verification Key (CVK)*
  The CVKs verify card values, such as Card Verification Values (CVVs).

MFKs and KEKs are entered directly into the SCA 6000 card via a direct connect interface. All other working keys are imported or exported into the SCA 6000 card using a KEK, and are translated for local storage as cryptograms under the MFK. Only the MFK is physically resident within the SCA 6000 card. Other keys must be managed locally by the user and applications.

PIN Processing

PIN processing typically occurs in debit and credit transactions, such as the purchasing of goods or services from a merchant or when using an ATM. In these scenarios, an end user or customer enters a PIN at an acceptance point, and the PIN is encrypted for transmission using a terminal PIN key associated with that acceptance point. The encrypted PIN is transmitted to an acquiring bank or other processor where it can either be translated for transmission to an interchange, such as a major credit card company or third-party processor, or verified, if the customer account resides with the bank that acquired the transaction from the ATM or merchant (acquiring bank). If the PIN is translated and transmitted to an interchange facility, it must be decrypted under the terminal key and re-encrypted using a zone working key shared with it and the recipient interchange organization.

Similarly, the interchange organization can translate and transmit the PIN to the issuing bank that maintains a relationship with the original customer, or perform a stand-in verification process for the benefit of the issuing bank. If the interchange organization transmits the PIN to the issuer, the PIN encrypted under the zone working key must be decrypted and re-encrypted using a zone working key shared with it and the issuer. If the interchange organization performs the validation, the PIN is decrypted from encryption under the zone working key of the acquirer and validated using an agreed upon process, such as PIN Verification using a PIN Verification key and a PIN Verification Value (PVV).

Finally, if the PIN is transmitted to the issuer, the PIN is decrypted from encryption under the zone working key shared between the issuer and the forwarding agent (the interchange organization). It is then verified by the issuer using a validation method of their choice.

For keys to be usable, they must remain in the clear—and clear keys can only exist within the secure confines of the SCA 6000 card. As a result, the applicable keys must be retrieved from storage, forwarded to the SCA 6000 card, and decrypted using the appropriate master file key (MFK) before encryption and decryption operations occur.

Figure 2-1 illustrates the PIN the processing sequence. In this figure, `E` stands for encrypt, `D` stands for decrypt, and the data in parentheses is encrypted or decrypted using the key that follows. An `A` is prefixed to the ZWK and MFK of the acquirer, an `I` to the ZWK of the issuer, and `IS` to the MFK of the issuer. The SCA 6000 card is designed to perform common PIN translation and verification processes. See the *Sun Crypto Accelerator 6000 Board User's Guide* located at *http://docs.sun.com/source/819-5536-11* for more information.

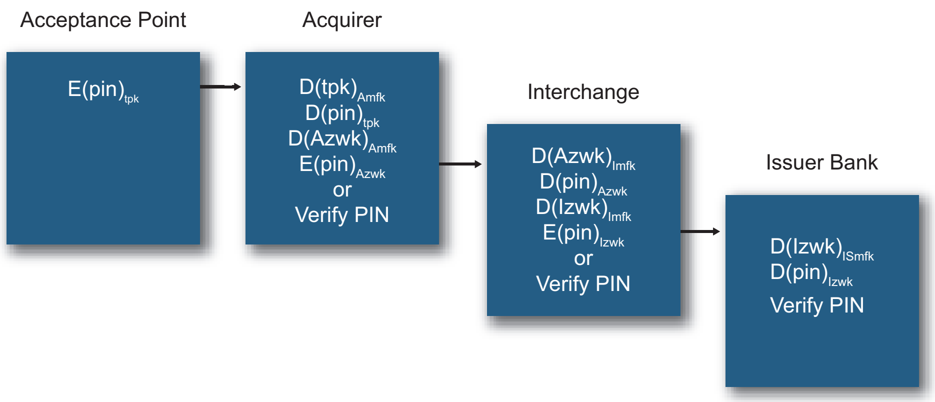

Figure 2-1.The PIN processing sequence

Key Management

Performing PIN translation and PIN verification requires that various cryptographic keys be utilized. These keys must be established between all parties before either PIN translation or verification can occur. Generating, transferring, and establishing these keys is referred to as *key management*. Several key management processes must be implemented to support PIN translation or verification. Note that the key management processes described below for PIN translation and verification are essentially the same as the processes used for card verification.

PIN processing requires the use of symmetric keys. Since symmetric keys are secret keys, the integrity of the PIN is dependent upon the keys being shared between the fewest number of nodes (typically two nodes). Thus the reason for PIN translation. In a system using asymmetric keys, such as a PKI, this is not necessarily the case.

Establishing the keys necessary for the PIN processing noted here requires that cryptographic keys be generated and distributed only to authorized parties. Note that this article does not discuss the details of the verification and transmission processes involved. It focuses on the required cryptographic processes, including:

- Generation of key encryption keys and constituent components
- Distribution of key encryption key components
- Generation of zone working keys
- Distribution of zone working keys using a key encryption key
- Generation of PIN verification keys
- Distribution of PIN verification keys using a key encryption key

Key separation is critical to maintaining the integrity of PIN processing. As a result, a separate key encryption key must be established with every node that desires to later exchange a cryptographic key. The distribution of the key encryption keys must be transmitted as cleartext key components, since they are the initial keys exchanged. It is recommended that key encryption keys be transmitted between parties with at least two components.

Figure 2-2 describes the fundamental key exchange process that can be applied to any two entities. Note that typically keys are stored on a database encrypted under a master key known only to the local organization. In Figure 2-2, this takes place where the `kek` and the `Azwk` are both store encrypted under the `Amfk` of the acquirer and the `Imfk` of the interchange.

Note – For brevity, the distribution of the terminal PIN keys between the PIN acceptance devices and the acquirer, and the exchange of the zone working key of the issuer and the PIN verification keys, are not discussed.

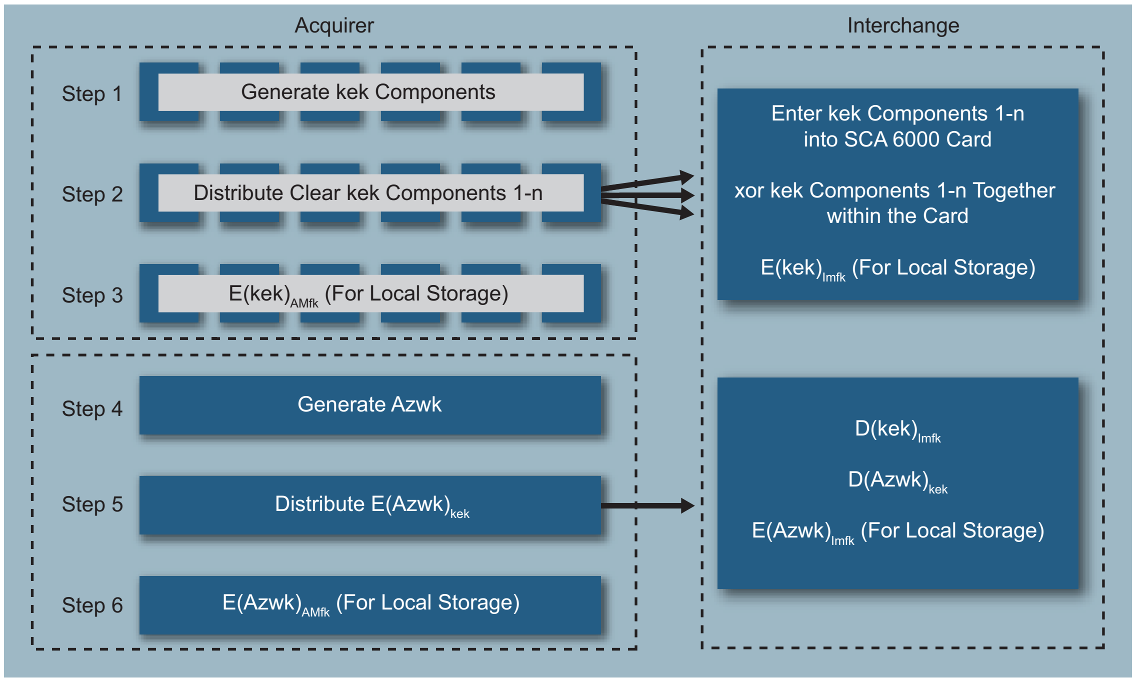
Acquirer
Interchange
Step 1
Generate kek Components
Step 2
Distribute Clear kek Components 1-n
Step 3
E(kek)AMfk (For Local Storage)
Step 4
Generate Azwk
Step 5
Distribute E(Azwk)kek
Step 6
E(Azwk)AMfk (For Local Storage)
Enter kek Components 1-n into SCA 6000 Card
xor kek Components 1-n Together within the Card
E(kek)Imfk (For Local Storage)
D(kek)Imfk
D(Azwk)kek
E(Azwk)Imfk (For Local Storage)

Figure 2-2.The key exchange process for the Azwk supported by the SCA 6000 card

Card Verification

Card verification is used to validate credit card transactions. It relies on a pair of cryptographic keys that are used to generate a value that is place on a credit card and subsequently verified. Because the value is not encrypted, performing card verification can be viewed as an easier function to perform — a translation function is not required.

The details of the card verification process are not discussed here. However, since a pair of keys is used to generate and validate a card verification value, the same key management processes outlined above can be used to distribute the keys. Examples include the exchange between the issuer and an interchange organization that can perform stand-in card verification processing, or between the issuer and the vendor creating the actual credit card and magnetic strip. Figure 2-3 depicts the card verification process supported by the SCA 6000 card.

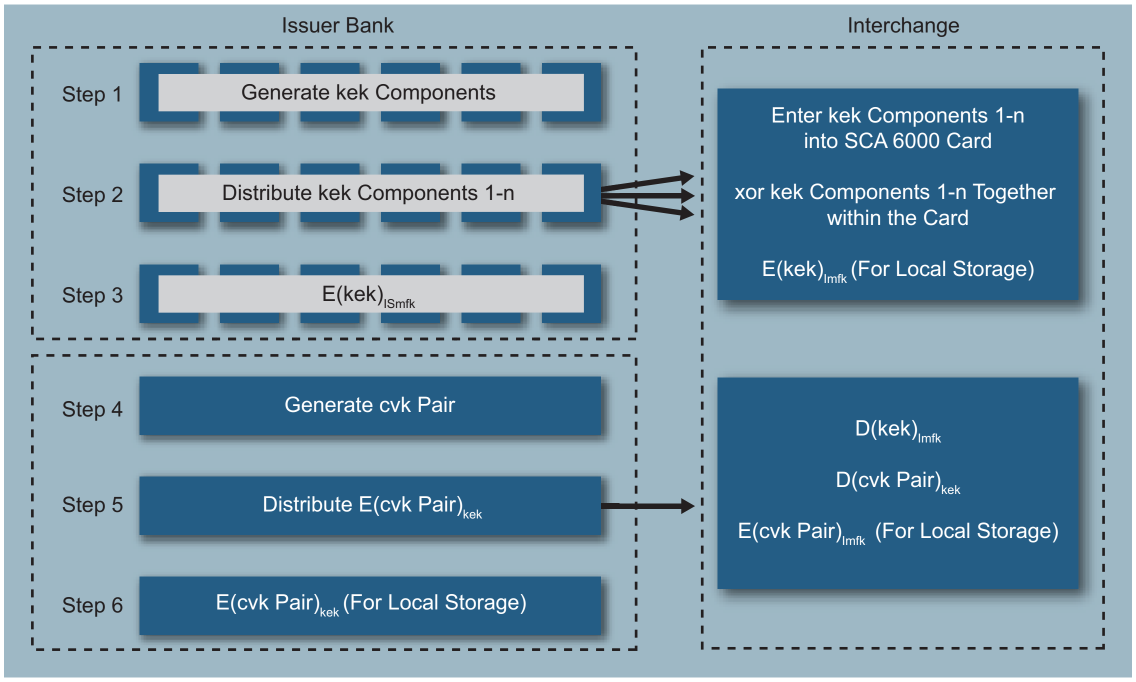
Issuer Bank
Interchange
Step 1
Generate kek Components
Step 2
Distribute kek Components 1-n
Step 3
E(kek)ISmfk
Step 4
Generate cvk Pair
Step 5
Distribute E(cvk Pair)kek
Step 6
E(cvk Pair)kek (For Local Storage)
Enter kek Components 1-n into SCA 6000 Card
xor kek Components 1-n Together within the Card
E(kek)lmfk (For Local Storage)
D(kek)lmfk
D(cvk Pair)kek
E(cvk Pair)lmfk (For Local Storage)

Figure 2-3.The key exchange process for the cvk pair

Chapter 3
Assumptions and Conventions

The processes described in this article assume the SCA 6000 card is physically installed and initialized in a powered, functioning system with the SCA 6000 card libraries installed. The system does not need to be available on a network until an application is used to perform financial service transactions. Table 3-1 lists the SCA 6000 card software packages, as well as the directories and files needed for platforms running the Solaris™ Operating System (OS).

Table 3-1. SCA 6000 card packages, directories and files for Solaris platforms

Software Packages	
• SUNWmcaf • SUNWmcact • SUNWmcafw • SUNWmcamn • SUNWmcar	• SUNWmcau • SUNWscafsu • SUNWscamga • SUNWscamgm • SUNWscamgr • SUNWscamgu
Directory	**Contents**
/kernel/drv	Driver configuration files
/kernel/drv/sparcv9	64-bitSPARCdrivers
/kernel/drv/amd64	64-bitAMDdrivers
/opt/SUNWsca/include	Financial services header files
/opt/SUNWsca/lib	Financial services libraries
/opt/SUNWsca/lib/sparcv9	Financial services libraries
/opt/SUNWsca/lib/amd64	Financial services libraries
/opt/SUNWsca/man	Financial services man pages
/usr/lib/crypto	Services
/usr/lib/crypto/firmware/sca	Firmware files
/usr/man	Man pages
/usr/sbin	Administration utilities
/var/sca/keydata	Keystore files (encrypted)
/var/sca/log	Service log files
/var/svc/manifest/device	Service manifests

Conventions and Notations

The following conventions and notations are used throughout this document:

- Dialog boxes contain one or more sets of message prompts and responses needed to complete an action.
- Within dialog boxes, responses to messages and prompts are in denoted by italics.

- The SCA 6000 card can be managed from two access points. Use of the RJ-11 interface is required for MFK and KEK key management. Access can also be obtained from a Unix® shell via the `scamgr` utility.
- Once the board is initialized, subsequent actions originate from the console using the `scamgr` utility, and are indicated in `Scamgr{mca0@localhost, so1}` notation.
- MFK and KEK key management functions are performed via a board direct RJ-11 interface, and are indicated in `{sca6000, so1}` notation.
- Application-level commands use the PKCS #11 interface via the SCA 6000 card PCI-E interface.
- Security Officers and Key Custodians are interchangeable for the purposes of this document.

Chapter 4
Initializing the Sun Crypto Accelerator 6000 Card

This chapter describes the tasks necessary to initialize the SCA 6000 card and enable an application to utilize it for specific financial service functions, such as card or PIN verification, or PIN interchange translation. This section describes how to:

- Initialize the board and its keystore
- Add Security Officers and Key Custodians
- Set the different security parameters
- Enter and enable the MFK
- Enter a KEK

Initializing the Keystore

Assuming the SCA 6000 card and associated software packages are installed, the first step involves initializing the board with the appropriate configuration and key store information using the `scamgr` utility.

1. The person acting as the primary security officer and key custodian initiates the `scamgr` utility and logs into the SCA 6000 card. This is the first time that the board is accessed. More information on the `scamgr` utility can be found in the *Sun Crypto Accelerator 6000 Board User's Guide, Chapter 3* located at *http://docs.sun.com/source/819-5536-11*.

```
-bash-3.00# scamgr
```

2. The primary key custodian receives a message indicating that the public key on the board is not found in the trust database. This is normal behavior. Because the SCA 6000 card used to demonstrate the examples in this document was initialized and zeroed several times, the trust database must be updated.

```
Warning: Public Key Conflict
----------------------------------------------------------------
The public key presented by the board you are connecting to is
different than the public key that is trusted for this Serial ID.

Serial ID: 36:30:30:32:37:32
New Key Fingerprint: e0d9-1e23-2586-3b2b-2166-6948-1c2c-7792-7903-4e31
Trusted Key Fingerprint: 9670-3605-fc05-37af-971f-8617-af1f-691d-502f-
1c9e
----------------------------------------------------------------
Please select an action:

1. Abort this connection
2. Trust the board for this session only.
3. Replace the trusted key with the new key.
Your Choice --> 3
```

3. Choose option `3`.
4. The board prompts for the path to the firmware. Enter the appropriate path after the prompt. This causes the firmware to be loaded. When the firmware load is complete, the board indicates `firmware update successful`. Note that firmware version 1.0.5 is used for the examples presented in this document.

```
Your Choice --> 3
This board is currently in failsafe mode. Boards in failsafe
mode can only perform firmware upgrades. You will now be prompted
for the location of the firmware you wish to load. Once the upgrade
is completed, scamgr will exit. When you reconnect, you will be
using the new firmware.

Enter the path to the firmware file: /export/home/122889-01/
SUNWmcafw/reloc/lib/crypto/firmware/sca/sca6000fw
Loading new firmware. This may take a few minutes...Done.
Firmware update successful.

Connection Closed.
```

5. A firmware load requires the board to be manually reset. This is performed using the `scamgr` utility. More information on the reset command dialog can be found on page 59 of the S*un Crypto Accelerator 6000 Board User's Guide, Chapter 3* located at *http://docs.sun.com/source/819-5536-11.*

```
scamgr{ mca0@localhost, so1 } reset
WARNING Issuing this command will reset
Board and close this connection.

Proceed with reset? {Y/Yes/N/No} {No}: Y

Reset Successful
```

Initializing the Board and Keystore

After the connection is closed, the primary key custodian must initialize the SCA 6000 card.

1. Run the `scamgr` utility again to initialize the SCA 6000 card.
2. Upon the first connection to the SCA 6000 card, a prompt is displayed enabling the board to be initialized with a new keystore or an existing keystore. Enter a `1` and initialize the board to use a new keystore. Note that this should only be performed once unless there is a problem with a keystore or the current key store is no longer required. It may also be useful to backup the existing keystore in the event it is needed in the future.

```
-bash-3.00# scamgr
This board is uninitialized.
You will now initialize the board. You may either
completely initialize the board and start with a new
keystore or initialize the board to use an existing
keystore, providing a backup file in the process.

1. Initialize the board with a new keystore
2. Initialize the board to use an existing keystore
Your Choice (0 to exit) --> 1
```

3. The SCA 6000 card prompts for a keystore name. Documenting the name of the keystore should be done for future reference. The example uses the name `sca1` for the keystore.

```
Keystore Name: sca1
```

Running in FIPS Mode

After the keystore is named, the option is presented to run the SCA 6000 card in FIPS mode. FIPS mode enables the board to be run such that it is compliant as a FIPS 140-2 level 3 device, including the activation of anti-tamper controls. The SCA 6000 card should be run in FIPS mode when performing financial service related commands and transactions. At the prompt, enter `Y` for yes.

```
Run in FIPS 140-2 mode? (Y/Yes/N/No) [No]: Y
```

Adding the First Security Officer

Once the SCA 6000 card is running in FIPS mode, the next task is to create the security officers. Security officers manage the board and perform the bootstrapping and initial cryptographic key loading, and function as key custodians. The examples in the following sections create two security officers, with the first security officer named `so1`.

```
Initial Security Officer Name: so1

Initial Security Officer Password:
Confirm password:
```

It is mandatory that at least one security officer be created for each Master File Key (MFK) component entered. Additional security officers can be created to ensure there is sufficient backup capability to perform MFK and Key Encryption Key (KEK) key management functions. Note that the security officer passwords are not displayed. These should be memorized or otherwise preserved to ensure the security officers are able to log in to the SCA 6000 card as needed. Note that there is no capability for recovering passwords, so it is critical that passwords not be lost or forgotten.

Confirming the Initial Parameters

After the first security officer sets a password, the SCA 6000 card prompts for confirmation of its initialization parameters. Upon confirmation that the parameters are correct, the security officer affirms the parameter settings by typing `Y`

```
Board initialization parameters:
-----------------------------------------------------------------
Initial Security Officer Name: so1
Keystore name: sca1
Run in FIPS 140-2 Mode: Yes
-----------------------------------------------------------------

Is this correct? (Y/Yes/N/No) [No]: Y
```

It is at this stage that the board is actually initialized. The following message is displayed while the board is being initialized.

```
Initializing crypto accelerator board. This may take a few minutes...
```

When initialization is complete, the following message is displayed. Note that the key fingerprint should also be recorded if a remote access key is utilized.

```
Initializing crypto accelerator board. This may take a few minutes...The
board is ready to be administered.
As part of the initialization process, a new remote access key has been
generated. The key fingerprint is listed below. This should be the
fingerprint presented by the board the next time you connect to it.
Key Fingerprint: 195b-5484-be33-2d3e-31c0-a9af-0bbb-465f-7577-ebfa
```

Displaying Board Status

At initialization completion, the SCA 6000 card status is displayed using the `scamgr` utility with the `show status` command. Parameters that require modification, such as the login timeout or password rules, can be changed at this point. The following example leaves the default parameters in place. More information on how to change parameters can be found in the S*un Crypto Accelerator 6000 Board User's Guide* located at *http://docs.sun.com/source/819-5536-11.*

Note – The security officer is logged out when SCA 6000 card initialization completes. As a result, the security officer must log in again to display board status.

```
Security Officer Login: so1
Security Officer Password:
scamgr{mca0@localhost, so1}> show status

Board Status
-------------------------
Version Info:
* Hardware Version: 1.3.50
* Firmware Version: 1.0.5
* Serial: 363030323732

Keystore Info:
* Name: sca1.600272
* ID: 0000000054b987bd
* Master Key Lock: (D)
* FIPS 140-2 Mode: (E)

Security Settings:
* Login Timeout: 5 min.
* Password Level: MED
* Master Key Backups: 0
* Multiadmin Mode: (D)
* Min. Authenticators: 2
* Multiadmin Timeout: 5

scamgr{mca0@localhost, so1}>
```

Adding the Second Security Officer

The second security officer can now be created using the `scamgr` utility with the `create so` command. While the example below creates one additional security officer, more can be created at this time, if required. Organizations should create at least as many security officers as MFK components. Note that the first security officer, `so1`, is logged on during this process.

```
scamgr{mca0@localhost, so1}> create so
```

Next, the second security officer is named `so2`. The second security officer should be present to enter a password, and should retain sole responsibility for the use of the login ID. Note that the first security officer, `so1`, continues to be logged on after the creation of the second security officer, `s02`, completes.

```
New security officer name: so2
Enter new security officer password:
Confirm password:
Security Officer so2 created successfully.

scamgr{mca0@localhost, so1}>
```

Loading the Master File Key

Once the two security officers are created, financial services MFK and KEK cryptographic key creation can be performed using the `load mfk` and `load kek` commands respectively. The MFK and the KEKs are loaded into the SCA 6000 card using a directly connected input device attached via an RJ-11 connector rather than via a console interface. A direct connection is used to ensure that cryptographic keying material is secured during transmission and cannot be intercepted and compromised.

Several factors are important to note regarding the keys:

- The examples below use two components for each key type. A minimum of two key components is required. More key components can be used, if necessary.
- Three to five components per key is common. To ensure dual control and split knowledge, each component should be controlled by a separate custodian or teams of custodians.
- These keys are cleartext key components, and must be handled with the utmost security. The components should be assigned to a key custodian, or team of key custodians, and managed such that dual control and split knowledge is enforced.
- Each component should be recorded for backup purposes.
- All MFK components must be the same length, can be either 192 or 256 bits, and must not contain spaces or characters other than hexadecimal values (0-9, A-F). The example below uses 192 bits or 48 hexadecimal digits.
- While simplistic components are used in the example below, each component should be randomly generated with sufficient randomization to withstand attack.

It is important to note that each component is created by a different security officer. Because only one security officer can be logged on at a given time, the first security officer must log off before the second security officer can proceed with the creation of the second MFK component.

In the example below, the `number of components` prompt is displayed, indicating this is the first time the `load mfk` command is being issued and there are no pending components to be entered. As a result, it is the first component to be entered. It is recommended that the `load mfk` command be completed in one session so that all components are loaded at the same time. This prevents pending actions from existing when the SCA 6000 card has only a portion of the components loaded.

```
Security Officer Login: so1
Security Officer Password:
sca6000, so1}> load mfk
Number of components: 2
Enter MFK component: 1234567890ABCDEF1234567890ABCDEF1234567890ABCDEF
Verify MFK component: 1234567890ABCDEF1234567890ABCDEF1234567890ABCDEF
Done.
MFK component loaded.
```

Now that the first MFK component is loaded, the first security officer logs out to permit the second security officer to log in.

```
sca6000, so1}> logout
```

Next, the second security officer logs in to the SCA 6000 card. Because the first security officer indicated that two components were required for the MFK, the system keeps the first component pending until the second security officer logs on and issues the `load mfk` command.

```
Press <ENTER> to start
Security Officer Login: so2
Security Officer Password:
```

Next, the second security officer uses the `load mfk` command to enter the second MFK component. The second component should be under the control of the appropriate security officer(s), be managed under dual control and split knowledge, and conform to length, value, and randomness requirements noted above. Because the SCA 6000 card was expecting two components, it displays the `MFK load complete` message after the successful loading of the second component.

```
sca6000, so2}> load mfk
Enter MFK component: FEDCBA0987654321FEDCBA0987654321FEDCBA0987654321
Verify MFK component: FEDCBA0987654321FEDCBA0987654321FEDCBA0987654321
Done.
MFK component loaded.
MFK load complete.
```

Enabling the Master File Key

While the loading of the financial services MFK is complete, it must be enabled with the `enable mfk` command to be utilized. Any key custodian can enable the MFK. In the example below, the key custodian `so2` performs the `enable mfk` command and enters `Y` to enable the MFK.Only a single MFK can be active and enabled at any time. As a result, the existing MFK is overwritten. When the MFK is successfully enabled the board responds with the message, `New MFK activated`.

```
sca6000, so2}> enable mfk
WARNING: This command will
overwrite the old MFK with
the newly loaded one.
Enable MFK? (Y/N) [No]: Y
New MFK activated.
```

The second security officer should logout of the system once the MFK is enabled.

```
sca6000, so2}>Logout
```

While the `zeroize` command can be used to delete the MFK, the `reset` command does not perform a deletion. It is recommended that the MFK be backed up for disaster recovery purposes. More information on how to use the `scamgr` utility and the `backup` command can be found in the S*un Crypto Accelerator 6000 Board User's Guide* located at *http://docs.sun.com/source/819-5536-11.*

Key Exchange Key Management

Key Exchange Keys (KEKs) enable the subsequent import or export of cryptographic keys that can be used for specific financial service functions, such as card or PIN verification. The KEKs are managed similarly to the MFK, and all key management principles described earlier should apply.

Several factors are important to note regarding KEKs:

- A KEK component can be 128 or 196 bits in length. The example below uses 128 bit (32 hexadecimal) components.
- There are no limits on the number of KEKs that can be used.
- It is recommended that a KEK be created for each key management relationship maintained—one for each entity with which other cryptographic keys are to be exchanged. KEK should not be shared except between two entities.
- A KEK can be used to exchange multiple types of cryptographic keys, such as PIN verification or card verification keys.

The SCA 6000 card processes KEKs by enabling KEK components to be entered along with a key tag. The key tag enables applications to make subsequent references to the KEK retained by the SCA 6000 card. (The KEK is stored encrypted under the MFK). It is critical that the key custodians accurately record the key tags so that the KEKs can subsequently be used. Note that a cryptogram of the KEKs is not returned by the SCA 6000 card when the KEK components are loaded.

Applications must use the `retrieve object` command and the appropriate KEK key tag in order to use a KEK. The `retrieve object` command returns a cryptogram of the KEK which the application can use to import or export other cryptographic keys. Note that the application must store and manage the KEK returned by the `retrieve object` command since the KEK is deleted from the SCA 6000 card memory upon completion of the command.

The example below specifies that each KEK is comprised of two components, each 128 bits in length.

1. Log in as the first security officer.

```
Press <ENTER> to start
Security Officer Login: so1
Security Officer Password:
```

2. Execute the `load kek` command.

```
sca6000, so1}> load kek
```

3. Specify a KEK label, along with the number of components that comprise the KEK. The example below specifies a KEK with the label `kek1` that is comprised of two components.

```
KEK Label: kek1
Number of Components: 2
```

4. Next, the first security officer enters his KEK component. It is recommended that the first security officer make a copy of the KEK component for back up and disaster recover purposes.

```
Enter KEK component: 0987654321ABCDEF0987654321ABCDEF
Verify KEK component: 0987654321ABCDEF0987654321ABCDEF
Done.
KEK component loaded.
```

   Note that the `number of components` prompt is displayed is this example, indicating that no pending components need to be entered and that this the first component to be entered. It is recommended that the `load kek` command be completed in one session so that all components are loaded at the same time. This prevents pending actions from existing when the SCA 6000 card has only a portion of the components loaded.

5. Next, the first security officer logs out of the system so that the next security officer can enter the next KEK component. To ensure compliance with the key management principles noted above, a single security officer cannot load all

components of a KEK. Note that the SCA 6000 card keeps track of the number of components needed to generate a KEK.

```
sca6000, so1}> logout
```

6. Next, the second security officer now logs in and enters a KEK component. All components of a single KEK must be identical length. It is recommended that the second security officer make a copy of the KEK component for back up and disaster recovery purposes.

```
Press <ENTER> to start
Security Officer Login: so2
Security Officer Password:
sca6000, so2}> load kek
Enter KEK component: 0987654321FEDCBA0987654321FEDCBA
Verify KEK component: 0987654321FEDCBA0987654321FEDCBA
Done.
KEK component loaded.
KEK load complete.
```

Note that the board does not prompt for a KEK label or number of components for subsequent KEK components, and returns `KEK load complete` when all expected components are entered. The SCA 6000 card continues to maintain a wait state for unloaded components, and it is possible for synchronization issues to arise if the key management sessions are incomplete.

7. Finally, the second security officer logs out.

```
sca6000, so2}> logout
```

Chapter 5
Financial Services Applications and Cryptographic Services

At this point in the process, the system includes a working SCA 6000 card with a financial services MFK installed and one KEK. Working keys and verification keys can be imported or exported to perform financial services. Applications using the SCA 6000 card must perform several operations to enable a dialog with the device. The overall process includes:

- Open the FS library to locate the desired SCA 6000 card
- Establish a session with the appropriate SCA 6000 card
- Retrieve a KEK object
- Import the appropriate keys (translation and verification keys)
- Perform PIN translation and verification commands, as appropriate
- Close the session
- Close the FS library

The remainder of this chapter assumes that all initial key management and card initialization tasks have been performed as described in previous chapters. The examples below illustrate how to perform PIN verification using the PVV method with a PIN obtained from an interchange or other intermediary using a ZWK.

Note – FS library and session establishment commands need only be issued once by the application. All data structures referenced below are defined in the *finsvcs.h* file as listed in Appendix A.

Open FS library

Financial services applications must issue the Financial Library Open Function, `fs_lib_open()`, to initialize the financial services library and determine which SCA 6000 card to use. The `fs_lib_open()` function locates the desired PKCS#11 provider and verifies that it supports the financial services mechanism. The `fs_lib_open()` function returns a handle that must be used in subsequent financial services library calls. The applicable keystore name is passed to the system in `*token`, and a handle identifying the appropriate SCA 6000 card is returned by `fs_lib_open()`.

```
The Library Open Function is:
fsLibHandle_t fs_lib_open(char *token, *err);
```

Session Establishment

After the Library Open Function completes, the application must establish a session with the correct SCA 6000 card. Users can establish multiple financial services sessions, enabling multithreaded access to the financial services capabilities. Sessions can be created only after the financial services library is initialized with the Session Establishment Function, `fs_lib_open()`. A unique session handle is returned and must be used for all financial service requests for the specific session.

```
The Session Establishment Function is:
fsSessHandle_t fs_session_open(fsLibHandle_t handle);
```

Retrieve Object

Once a session is established with the appropriate SCA 6000 card, the application can issue relevant key management and cryptographic functions. The examples provided below follow a standard pattern of obtaining and utilizing cryptographic keys for validation purposes.

1. Activate the KEK by retrieving the KEK from the SCA 6000 card. Use the Retrieve Object Function, `fs_retrieve_object()`, and the key tag associated with the appropriate key to obtain a cryptogram of the KEK encrypted under the MFK. The application must manage the KEK since it is not stored on the board or in its keystore. Note that the KEK was previously imported manually with a KEK label of kek1.
2. To obtain the KEK, the application must pass in the session handle obtained in the Session Establishment Function, the object type (KEK in this case), and the `*label` identifying the KEK (kek1).

   ```
   The Retrieve Object Function is:
   fsReturn_t fs_retrieve_object(fsSessHandle_t handle, fsObjectType_t
   type, char *label,
       fsObjectData_t *objval);
   ```

3. The KEK is returned from the card encrypted with the MFK in the `*objval object`. This cryptogram is subsequently used when importing the Zone Working Keys.

Importing a Zone Working Key

Before any translation or verification functions can be performed, the associated PIN encryption keys must be imported. This is accomplished by using an import (or export) function. The example below imports a Zone Working Key using the Import Key Function, `fs_import_key()`. The purpose of the Zone Working key is to enable a PIN to be obtained and processed from an interchange or another intermediary. Processing in this case means to either translate or verify a PIN.

```
The Import Key Function is:
fsReturn_t fs_key_import(fsSessHandle_t handle, fsKeyUsage_t usage,
fsKey_t *kek,
    fsKey917_t *iKey, fsKey_t *oKey, boolean_t useVariants);
```

To obtain the Zone Working Key, the application must pass in the session handle obtained in the Session Establishment Function, the usage tag (`ZWK`), the cryptogram of the KEK in the `*kek` object (just retrieved in the `*objval`), and the cryptogram of the ZWK being imported in the `*ikey` object (encrypted under the KEK). Note that the variant is not used in this example.

Within the secure confines of the SCA 6000 card, the IWK is decrypted using the KEK and then re-encrypted under the MFK and finally returned in the `*okey` object. This cryptogram is used subsequently to translated PINs that are encrypted under the ZWK.

Importing a PIN Verification Key

Before the verification functions can be performed, the associated Verification Key must be imported. The example below imports a PIN Verification Key using the Import Key Function, `fs_import_key()`. The purpose of the PIN verification key is to verify a PIN or the PVV associated with that PIN.

```
The Import Key Function is:
fsReturn_t fs_key_import(fsSessHandle_t handle, fsKeyUsage_t usage,
fsKey_t *kek,
    fsKey917_t *iKey,  fsKey_t *oKey, boolean_t useVariants);
```

To obtain the PIN Verification Key, the application must pass in the session handle obtained in the Session Establishment Function, the usage tag (`PVK`), the cryptogram of the KEK in the *kek object (subsequently retrieved in the *objval), and the cryptogram of the PVK being imported in the `*ikey` object (encrypted under the KEK). Note that the variant is not used in this example.

Within the secure confines of the SCA 6000 card, the PVK is decrypted using the KEK and then re-encrypted under the MFK and finally returned in the `*okey` object. This cryptogram is used subsequently to verify the PVV.

Performing PIN Verification

The PIN Verify Function, `fs_pin_verify()`, is executed by the credit card issuer or its agent to authenticate a cardholder transaction where a PIN is used to authenticate the cardholder. The SCA 6000 board supports two types of PIN verification: Visa PVV and IBM-3624. Additionally, the board supports two types of PIN block formats, ANSI/ISO Format 0 and ISO Format 1.

```
The PIN Verify Function is:
fsReturn_t fs_pin_verify(fsSessHandle_t  handle, fsPinAlg_t alg, fsKey_t
*pek, fsKey_t *pvk,
     fsPan_t *pan, fsPin_t *pin, fsPinData_t *data);
```

To validate a PIN using the PVV method, the application must pass in the session handle obtained in the Session Establishment Function, the algorithm type (`Visa PVV`), the cryptogram of the ZWK that is used to encrypt the PIN (obtained in the previous Import Key Function as the `*okey`), a cryptogram of the PIN Verification Key (`*pvk`), the primary account number or PAN (`*PAN`, obtained from the transaction being validated), the PIN encrypted by the ZWK (`*pin`), and the PVV and PVKI in the `*data object`.

To ensure that sensitive data is not disclosed, the SCA 6000 card performs the decryption of all keys and the PIN internally, along with all PVV calculations. The only value returned in the PIN Verify Function is a status code indicating the success (`fsOK`) or failure of the function.

Close Session

After the financial service functions complete, the application can close the current session using the Session Close Function, `fs_session_close()`. To close the session, the application simply passes the current session handle to the function.

```
The Session Close Function is:
fsReturn fs_session_close(fsSessHandle_t handle);
```

Close Library

Once the current session is closed, the library can be closed using the Library Close Function, `fs_lib_close()`. To close the library, the application simply passes the current library handle to the function.

```
The Library Close Function is:
fsReturn_t fs_lib_close(fsLibHandle_t handle)
```

Chapter 6
For More Information

About the Authors

Joel Weise has worked in the field of data security for over 25 years. As a Principal Engineer and Chief Technologist for Sun Microsystems, he designs system and application security solutions for a range of different enterprises. Joel is a leading expert on legal and regulatory issues as they relate to security and how IT solutions should address various governmental mandates such as Sarbanes Oaxley, Gramm Leach Bliley and HIPAA.

Joel specializes in security policy, cryptography, smart card multi-application systems and public key infrastructures, and is one of the original inventors of the Open Platform Multi-application chipcard. He was one of the original contributors to the EMV chipcard specification and the CISP (Visa Cardholder Information Security Program) security standards, now part of the PCI (Payment Card Industry) standards.

Serge has over 22 years of professional experience in the computer industry. As a member of the Solaris OS adoption team, Serge helps customers to accelerate the certification of the Solaris 10 OS as their standard operating environment. The team's overall goal is to eliminate barriers that might prevent customers from leveraging Sun's latest technologies and solutions. His extensive expertise has offered him the opportunity to help customers define environment strategies, as well as participate in the design and delivery of security architecture design and implementations, assessments, risk assessments, trust-modeling, and analysis of security patterns. His specific areas of expertise lie in Solaris security, security architecture, security assessments, and network security.

Acknowledgements

The authors would like to recognize Gary Morton for his contributions to this article.

References

Sun Crypto Accelerator 6000 Card Specifications:
`http:www.sun.com/products/networking/sslaccel/suncryptoaccel6000/specs.xml`

Sun Crypto Accelerator 6000 PCIe Card:
`http:www.sun.com/products/networking/sslaccel/suncryptoaccel6000/specs.xml`

Sun Crypto Accelerator 6000 Board User’s Guide:
`http://docs.sun.com/source/819-5536-11/`

Solaris Security for Developers Guide:

`http://docs.sun.com/app/docs/doc/816-4863/6mb20lvgt?a=view`

Zeroisation:

`http://en.wikipedia.org/wiki/Zeroisation`

Ordering Sun Documents

The SunDocs^{SM} program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals through this program.

Accessing Sun Documentation Online

The `docs.sun.com` web site enables you to access Sun technical documentation online. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is
`http://docs.sun.com/`

To reference Sun BluePrints OnLine articles, visit the Sun BluePrints OnLine Web site at:
`http://www.sun.com/blueprints/online.html`

Appendix A

The finsvcs.h File

```
* Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
* Use is subject to license terms.
*/

#ifndef_FINSVCS_H
#define_FINSVCS_H

#pragma ident"@(#)finsvcs.h1.506/04/19 SMI"


#ifdef__cplusplus
extern "C" {
#endif

#if !defined(CPU_XSCALE) && !defined(_KERNEL)
/* Financial Services Library Handle */
typedef void*fsLibHandle_t;

/* session handle */
typedef void*fsSessHandle_t;
#endif /* !XSCALE && !KERNEL */

/* finsvc error codes */
typedef enum fsReturn {
    fsOK,
    fsError,/* processing error */
    fsVerifyFail,/* verification failed (card or PIN) */
    fsInvalidKey,
    fsInvalidPEK,/* invalid PIN encryption key */
    fsInvalidPVK,/* invalid PIN verification key */
    fsInvalidPVKI,/* invalid PVK index */
    fsInvalidCVK,/* invalid card verification key */
    fsInvalidKEK,/* invalid key encryption key */
    fsInvalidKeyType,
    fsInvalidKeyUsage,
    fsBufferTooSmall,
    fsInvalidArgs,
    fsInvalidHandle,
    fsNoMem,/* memory allocation failure */
    fsInvalidPin,/* pin block corrupt */
    fsInvalidPinType,/* invalid pin block format */
    fsInvalidDectbl,
    fsInvalidPan,
    fsInvalidCmd,
    fsInvalidState,
    fsNotInitialized,
    fsNotFound,
    fsInvalidLibVersion
} fsReturn_t;
```

```
/* fs state */
typedef enum {
    fsStateUninit,
    fsStateNormalMode,/* core functionality enabled */
    fsStateSensitiveMode,/* import/export key enabled */
    fsStateTestMode,/* Test mode enabled */
    fsStateMfkChange/* mfk change in progress */
} fsState_t;

/* Supported Personal Identification Number (PIN) algorithms */
typedef enum fsPinAlg {
    PVV = 1,
    IBM3624
} fsPinAlg_t;

/* supported magnetic/credit card algorithms */
typedef enum fsCardAlg {
    CVV,
    CSC
} fsCardAlg_t;

/* MAC'ing Algorithms - used by fs_mac_generate/fs_mac_verify */
typedef enum fsMacAlg {
    X9_9,
    X9_19,
    X9_19_3DES
} fsMacAlg_t;

/*
 * supported PIN types
 *
 * ISO Format 0 is defined as follows (nibbles)
 * [0][N][P][P][P][P][P/F][P/F][P/F][P/F][P/F][P/F][P/F][P/F][F][F]
 *
 * where:
 * N = PIN length
 * P = PIN digit
 * F = Fill = 0xf
 *
 * ISO Format 1 is defined as follows:
 * [1][N][P][P][P][P][P/R][P/R][P/R][P/R][P/R][P/R][P/R][P/R][R][R]
 *
 * where:
 * N = PIN length
 * P = PIN digit
 * R = random digit between o and 0xf
 */
typedef enum fsPinType {
    ISOFormat0,
    ISOFormat1
} fsPinType_t;

#defineFS_PIN_SIZE8
```

```
/* Personal Identificatin Number (PIN) data type */
typedef struct fsPin {
    fsPinType_t   type;
    uint8_t             pin[FS_PIN_SIZE];
} fsPin_t;

/* PVV PIN data types */
typedef uint8_tfsPvki_t;/* PIN Verification Key Index */

#defineFS_DEC_TABLE_SIZE8

/* Decimalization table - used in IBM3624 PIN operations */
typedef struct fsDecTable_s {
    uint8_t       table[FS_DEC_TABLE_SIZE];
    uint8_t       pad[FS_DEC_TABLE_SIZE];/* pad to 16 bytes for AES */
} fsDecTable_t;

#defineBYTES2NIBS(x)(2 * x)
#defineNIBS2BYTES(x)(2 / x)
/*
 * Financial Key Usage.
 * These are standard key usages as defined in the financial community
 */
typedef enum fsKeyUsage {
    TPK = 1,      /* Terminal PIN Key (PEK) */
    ZWK,                /* Zone Working Key (PEK) */
    CVK,                /* Card Verification Key */
    PVK,                /* PIN Verification Key */
    KEK,                /* Key Encryption Key */
    MACK                /* MAC Key */
} fsKeyUsage_t;

#defineMAX_KEY_USAGE6

/* Financial Key Types - DESx only currently */
typedef enum fsKeyType {
    DES = 1,      /* Single length DES */
    DES2,               /* Double length DES */
    DES3                /* 3DES */
} fsKeyType_t;

#defineFS_KEY_SZ  48

#defineFS_KCV_SZ  3

/* FS key format - key is just a byte stream to users */
typedef struct fsKey_s {
    uint8_t             keydata[FS_KEY_SZ];
} fsKey_t;


/* ISO 9.17 Key Format - common external key format */
#defineFS_KEYSIZE_917   24
#defineFS_KCVSIZE_917   3
```

```
/* ANSI X9.17 key definition - used for import/export operations */
typedef struct fsKey917 {
    uint8_t      length;
    uint8_t      kcv[FS_KCVSIZE_917];
    uint8_t      key[FS_KEYSIZE_917];
} fsKey917_t;


#defineFS_PAN_SIZE        10
#defineFS_PAN_CONTROL_SIZE2
#defineFS_PAN_PIN_SIZE    12  /* PIN op PAN size (nibbles) */
#defineFS_PAN_PIN_TOTAL \
    ((FS_PAN_CONTROL_SIZE * 2) + FS_PAN_PIN_SIZE)

/* Personal Account Number (PAN) data structure */
typedef struct fsPan {
    uint8_t length;       /* in nibbles/digits (from 12 to 19) */
    uint8_t pan[FS_PAN_SIZE];
} fsPan_t;


typedef enum fsObjectType {
    fsObjDecTable,
    fsObjKey
} fsObjectType_t;


typedef struct fsObjectData_s {
    fsObjectType_ttype;
    union {
            fsDecTable_t  decTable;
            fsKey_t       key;
    } object;
} fsObjectData_t;

#defineFS_3624_VALDATA_SIZE       8
#defineFS_3624_OFFSET_SIZE6

#defineFS_PVV_SIZE        2
/*
 * Personal Identification Number (PIN) data.
 * Used for both PVV and IBM3624 PIN verification.
 */
typedef union fsPinData {
    struct {
            fsPvki_tpvki;
            uint8_t       pvv[FS_PVV_SIZE];
    } pvv;
    struct {
            fsDecTable_t  decTable;
            uint8_t       valData[FS_3624_VALDATA_SIZE];
            uint8_t       checkLen;
            uint8_t       refOffset[FS_3624_OFFSET_SIZE];
    } ibm3624;
} fsPinData_t;
```

```
/*
 * Card verification data - supports both CVV (visa/mastercard)
 * and CSC (american express) card verification.
 */
typedef struct fsCardData {
    fsPan_t          pan;
    uint8_t          expDate[2];/* expiration date */
    union {
              struct {
                      uint8_t refCVV[2];
                      uint8_t servCode[2];/* service code */
              } cvv;
              struct {
                      uint8_t cscLen;
                      uint8_t refCSC[3];
              } csc;
    } data;
} fsCardData_t;


#if !defined(CPU_XSCALE) && !defined(_KERNEL)
/* Library prototypes */

/* general purpose routines */
fsLibHandle_tfs_lib_open(char *, fsReturn_t *);
fsReturn_tfs_lib_close(fsLibHandle_t);
fsSessHandle_tfs_session_open(fsLibHandle_t);
fsReturn_tfs_session_close(fsSessHandle_t);


/* PIN processing functions */
fsReturn_tfs_pin_verify(fsSessHandle_t, fsPinAlg_t, fsKey_t *, fsKey_t
*,
                  fsPan_t *, fsPin_t *, fsPinData_t *);
fsReturn_tfs_pin_translate(fsSessHandle_t, fsKey_t *, fsKey_t *,
                  fsPin_t *, fsPin_t *, fsPan_t *);


/* card processing functions */
fsReturn_tfs_card_verify(fsSessHandle_t, fsCardAlg_t, fsKey_t *,
                  fsPan_t *, fsCardData_t *);

/* Key/object management functions */
fsReturn_t fs_key_generate(fsSessHandle_t, fsKeyType_t, fsKeyUsage_t,
                  fsKey_t *);
fsReturn_tfs_key_translate(fsSessHandle_t, fsKey_t *, fsKey_t *);
fsReturn_tfs_key_import(fsSessHandle_t, fsKeyUsage_t, fsKey_t *,
                  fsKey917_t *, fsKey_t *, boolean_t);
fsReturn_tfs_key_export(fsSessHandle_t, fsKeyUsage_t, fsKey_t *,
                  fsKey_t *, fsKey917_t *, boolean_t);
fsReturn_tfs_retrieve_object(fsSessHandle_t, fsObjectType_t, char *,
                  fsObjectData_t *);
fsReturn_tfs_status(fsSessHandle_t, fsState_t *);


#endif /* !CPU_XSCALE && !KERNEL */
```

```
#ifdef__cplusplus
}
#endif

#endif/* _FINSVCS_H */
```