Cisco Secure Access Control Server (ACS) FIPS module
Network Security Services (NSS) 3.12.5 and 3.12.5.1

FIPS 140-2 Non-Proprietary Security Policy Level 1 Validation

Version 0.3
March 15, 2013

Table of Contents
List of Tables 2
List of Figures 2
Introduction 2
Platform List 3
Note on Calling the API Functions 3
Security Rules 4
Authentication Policy 7
Specification of Roles 7
   Role-Based Authentication 8
   Strength of Authentication Mechanism 8
   Multiple Concurrent Operators 9
Access Control Policy 9
Introduction

A security policy includes the precise specification of the security rules under which the cryptographic module must operate, including rules derived from the security requirements of the FIPS PUB 140-2 standard, and the additional security rules listed below. The rules of operation of the cryptographic module that define within which role(s) and under what circumstances (when
performing which services) an operator is allowed to maintain or disclose each security relevant data item of the cryptographic module.

There are three major reasons for developing and following a precise cryptographic module security policy:

- To induce the cryptographic module vendor to think carefully and precisely about whom they want to allow access to the cryptographic module, the way different system elements can be accessed, and which system elements to protect.
- To provide a precise specification of the cryptographic security to allow individuals and organizations (e.g., validators) to determine whether the cryptographic module, as implemented, does obey (satisfy) a stated security policy.
- To describe to the cryptographic module user (organization, or individual operator) the capabilities, protections, and access rights they will have when using the cryptographic module.

The NSS cryptographic module is an open-source, general-purpose cryptographic library, with an API based on the industry standard PKCS #11 version 2.20 [1]. It is available for free under the Mozilla Public License, the GNU General Public License, and the GNU Lesser General Public License. The NSS cryptographic module is jointly developed by Red Hat and Sun engineers and is used in Mozilla Firefox, Thunderbird, and many server applications from Red Hat and Sun.

The NSS cryptographic module has two modes of operation: the FIPS Approved mode and non-FIPS Approved mode. By default, the module operates in the non-FIPS Approved mode. To operate the module in the FIPS Approved mode, an application must adhere to the security rules in the Security Rules section and initialize the module properly. If an application initializes the NSS cryptographic module by calling the standard PKCS #11 function C_GetFunctionList and calls the API functions via the function pointers in that list, it selects the non-FIPS Approved mode. To operate the NSS cryptographic module in the FIPS Approved mode, an application must call the API functions via an alternative set of function pointers. Rule 7 of the Security Rules section specifies how to do this.

This document may be freely reproduced and distributed in its entirety.

**Platform List**

FIPS 140-2 conformance testing of the NSS cryptographic module was performed on the following platforms listed below. The list was configured with no elliptic curve cryptography (ECC) support.

- Security Level 1
- Cisco CentOS (CARS)

**Note on Calling the API Functions**

The NSS cryptographic module has two parallel sets of API functions, FC_xxx and NSC_xxx, that implement the FIPS Approved and non-FIPS Approved modes of operation, respectively. For example, FC_Initialize initializes the module's library for the FIPS Approved mode of operation, whereas its counterpart NSC_Initialize initializes the library for the non-FIPS Approved mode of operation. All the API functions for the FIPS Approved mode of operation are listed in the Specification of Services section.

Among the module's API functions, only FC_GetFunctionList and NSC_GetFunctionList are exported and therefore callable by their names. (The C_GetFunctionList function mentioned in the Introduction section is also exported and is
just a synonym of NSC_GetFunctionList.) All the other API functions must be called via
the function pointers returned by FC_GetFunctionList or NSC_GetFunctionList. 
FC_GetFunctionList and NSC_GetFunctionList each return a
CK_FUNCTION_LIST structure containing function pointers named C_xxx such as
C.Initialize and C_Finalize. The C_xxx function pointers in the CK_FUNCTION_LIST
structure returned by FC_GetFunctionList point to the FC_xxx functions, whereas the
C_xxx function pointers in the CK_FUNCTION_LIST structure returned by
NSC_GetFunctionList point to the NSC_xxx functions.

For brevity, we use the following convention to describe API function calls. Again we use
FC_Initialize and NSC_Initialize as examples:

- When we say “call FC_Initialize,” we mean “call the
  FC_Initialize function via the C.Initialize function pointer
  in the CK_FUNCTION_LIST structure returned by
  FC_GetFunctionList.”

- When we say “call NSC_Initialize,” we mean “call the
  NSC_Initialize function via the C.Initialize function
  pointer in the CK_FUNCTION_LIST structure returned by
  NSC_GetFunctionList.”

Security Rules
The following list specifies the security rules that the NSS cryptographic module and each product
using the module must adhere to:

1. The NSS cryptographic module consists of software libraries compiled for each
   supported platform.

2. The cryptographic module relies on the underlying operating system to ensure the
   integrity of the cryptographic module loaded into memory.

3. Applications running in the FIPS Approved mode call FC_GetFunctionList for
   the list of function pointers and call the API functions via the function pointers in
   that list for all cryptographic operations. (See the Note on Calling the API
   functions section.) The module changes from FIPS Approved mode to non-FIPS
   Approved mode when a FC_Finalize/NSC_Initialize sequence is executed;
   it changes from non-FIPS Approved mode to FIPS Approved mode when a
   NSC_Finalize/FC_Initialize sequence is executed.

4. NSS cryptographic module can be configured to use different private key database
   formats: key3.db or key4.db. “key3.db” format is based on the Berkeley DataBase
   engine and should not be used by more than one process concurrently. “key4.db”
   format is based on SQL DataBase engine and can be used concurrently by multiple
   processes. Both databases are considered outside the cryptographic boundary. The
   interface code of the NSS cryptographic module that accesses data stored in the
   database is considered part of the cryptographic boundary as the interface code
   encrypts/decrypts data.

5. Secret and private keys, plaintext passwords, and other security-relevant data items are
   maintained under the control of the cryptographic module. Secret and private keys
   are only to be passed to the calling application in encrypted (wrapped) form with
   FC_WrapKey using Triple DES or AES (symmetric key algorithms) or RSA
   (asymmetric key algorithm). Note: If the secret and private keys passed to higher-
   level callers are encrypted using a symmetric key algorithm, the encryption key may
   be derived from a password. In such a case, they should be considered to be in
   plaintext form in the FIPS Approved mode.
6. Once the FIPS Approved mode of operation has been selected, the user must only use the FIPS 140-2 cipher suite.

7. The FIPS 140-2 cipher suites consist solely of:
   - Triple DES (FIPS 46-3) or AES (FIPS 197) for symmetric key encryption and decryption.
   - Secure Hash Standard (SHA-1, SHA-256, SHA-384, and SHA-512) (FIPS 180-2) for hashing.
   - HMAC (FIPS 198) for keyed hash.
   - Random number generator Hash DRBG (NIST SP800-90).
   - Diffie-Hellman primitives or RSA encrypt/decrypt may be used by user applications to implement approved key establishment methods. The module does not provide key establishment functionality.
   - DSA (FIPS 186-2 with Change Notice 1), RSA (PKCS #1 v2.1), for signature generation and verification.

<table>
<thead>
<tr>
<th><strong>Table 1 Algorithm Validation Certificates</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algorithm</strong></td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>DRBG</td>
</tr>
<tr>
<td>RSA</td>
</tr>
<tr>
<td>DSA</td>
</tr>
<tr>
<td>Triple DES</td>
</tr>
<tr>
<td>AES</td>
</tr>
<tr>
<td>SHS</td>
</tr>
<tr>
<td>HMAC</td>
</tr>
</tbody>
</table>

The NSS cryptographic module implements the following non-Approved algorithms, which must not be used in the FIPS Approved mode of operation:
   - RC2, RC4, DES, SEED, or CAMELLIA for symmetric key encryption and decryption
MD2 or MD5 for hashing

8. Once the FIPS Approved mode of operation has been selected, Triple DES and AES must be limited in their use to performing encryption and decryption using either ECB or CBC mode.

9. Once the FIPS Approved mode of operation has been selected, SHA-1, SHA-256, SHA-386, and SHA-512 must be the only algorithms used to perform one-way hashes of data.

10. Once the FIPS Approved mode of operation has been selected, RSA must be limited in its use to generating and verifying PKCS #1 signatures, and to encrypting and decrypting key material for key exchange.

11. Once the FIPS Approved mode of operation has been selected, DSA can be used in addition to RSA to generate and verify signatures.

12. The module does not share CSPs between an Approved mode of operation and a non-Approved mode of operation.

13. All cryptographic keys used in the FIPS Approved mode of operation must be generated in the FIPS Approved mode or imported while running in the FIPS Approved mode.

14. The cryptographic module performs explicit zeroization steps to clear the memory region previously occupied by a plaintext secret key, private key, or password. A plaintext secret or private key must be zeroized when it is passed to a FC_DestroyObject call. All plaintext secret and private keys must be zeroized when the NSS cryptographic module: is shut down (with a FC_Finalize call); or when reinitialized (with a FC_InitToken call); or when the state changes between the FIPS Approved mode and non-FIPS Approved mode (with a NSC_Finalize/FC_Initialize or FC_Finalize/NSC_Initialize sequence). All zeroization is to be performed by storing the value 0 into every byte of the memory region previously occupied by a plaintext secret key, private key, or password.

15. The environment variable NSS_ENABLE_AUDIT must be set to 1 before the application starts.

16. The NSS cryptographic module consists of the following shared libraries and the associated .chk files:
   - libsoftokn3.so
   - libsoftokn3.chk
   - libfreebl3.so
   - libfreebl3.chk
   - libnssdbm3.so
   - libnssdbm3.chk

The NSS cryptographic module requires the Netscape Portable Runtime (NSPR) libraries. NSPR provides a cross-platform API for non-GUI operating system facilities, such as threads, thread synchronization, normal file and network I/O, interval timing and calendar time, atomic operations, and shared library linking. NSPR also provides utility functions for strings, hash tables, and memory pools. NSPR is outside the cryptographic boundary because none of the NSPR functions are security-relevant. NSPR consists of the following shared libraries:
   - libplc4.so
The installation instructions are:

**Step 1:** Install the shared libraries and the associated .chk files in a directory on the shared library search path, which could be a system library directory (/usr/lib) or a directory specified in the following environment variable:

```
• LD_LIBRARY_PATH
```

**Step 2:** Use the chmod utility to set the file mode bits of the shared libraries to 0755 so that all users can execute the library files, but only the files' owner can modify (i.e., write, replace, and delete) the files. For example:

```bash
$ chmod 0755 libsoftokn3.so libfreebl*3.so libplc4.so libplds4.so libnspr4.so
```

The discretionary access control protects the binaries stored on disk from being tampered with.

**Step 3:** Use the chmod utility to set the file mode bits of the associated .chk files to 0644. For example:

```bash
$ chmod 0644 libsoftokn3.chk libfreebl*3.chk libnssdbm3.chk
```

**Step 4:** As specified in Rule 7, to operate the NSS cryptographic module in the FIPS Approved mode, an application must call the alternative PKCS #11 function FC_GetFunctionList and call the API functions via the function pointers in that list. The user must initialize the password when using the module for the first time. Before the user password is initialized, access to the module must be controlled. See the Sample Cryptographic Module Initialization Code section below for sample code.

(End of Security Rules)

## Authentication Policy

### Specification of Roles

The NSS cryptographic module supports two authorized roles for operators.

- The NSS User role provides access to all cryptographic and general-purpose services (except those that perform an installation function) and all keys stored in the private key database. An NSS User utilizes secure services and is also responsible for the retrieval, updating, and deletion of keys from the private key database.

- The Crypto Officer role is supported for the installation of the module. The Crypto Officer must control the access to the module both before and after installation. Control consists of management of physical access to the computer, executing the NSS cryptographic module code as well as management of the security facilities provided by the operating system. The NSS cryptographic module does not have a maintenance role.
Role-Based Authentication

The NSS cryptographic module uses role-based authentication to control access to the module. To perform sensitive services using the cryptographic module, an operator must log into the module and perform an authentication procedure using information unique to that operator (password). The password is initialized by the NSS User as part of module initialization. Role-based authentication is used to safeguard a user's private key information. However, discretionary access control is used to safeguard all other information (e.g., the public key certificate database).

If a function that requires authentication is called before the operator is authenticated, it returns the CKR_USER_NOT_LOGGED_IN error code. Call the FC_Login function to provide the required authentication.

A known password check string, encrypted with a Triple-DES key derived from the password, is stored in an encrypted form in the private key database (either key3.db or key4.db) in secondary storage. Note: This database lies outside the cryptographic boundary.

Once a password has been established for the NSS cryptographic module, the module allows the user to use the private services if and only if the user successfully authenticates to the module. Password establishment and authentication are required for the operation of the module. Password authentication does not imply that any of the roles are considered to be authorized for the purposes of Level 2 FIPS 140-2 validation.

In order to authenticate to the cryptographic module, the user enters the password, and the cryptographic module verifies that the password is correct by deriving a Triple-DES key from the password, using an extension of the PKCS #5 PBKDF1 key derivation function with an 16-octet salt, an iteration count of 1, and SHA-1 as the underlying hash function, decrypting the stored encrypted password check string with the Triple-DES key, and comparing the decrypted string with the known password check string.

The user's password acts as the key material to encrypt/decrypt secret and private keys. Note: Since password-based encryption such as PKCS #5 is not FIPS Approved, password-encrypted secret and private keys should be considered to be in plaintext form in the FIPS Approved mode. Secret and private keys are only stored in encrypted form (using a Triple-DES key derived from the password) in the private key database (key3.db/key4.db) in secondary storage. Note: Password-encrypted secret and private keys in the private key database should be considered to be in plaintext form in the FIPS Approved mode.

Strength of Authentication Mechanism

In the FIPS Approved mode, the NSS cryptographic module imposes the following requirements on the password. These requirements are enforced by the module on password initialization or change.

- The password must be at least seven characters long.
- The password must consist of characters from three or more character classes. We define five character classes: digits (0-9), ASCII lowercase letters, ASCII uppercase letters, ASCII non-alphanumeric characters (such as space and punctuation marks), and non-ASCII characters. If an ASCII uppercase letter is the first character of the password, the uppercase letter is not counted toward its character class. Similarly, if a digit is the last character of the password, the digit is not counted toward its character class.

To estimate the probability that a random guess of the password will succeed, we assume that:
- The characters of the password are independent with each other, and
- The probability of guessing an individual character of the password is less than 1/10.
Since the password is at least 7 characters long, the probability that a random guess of the password will succeed is less than \((1/10)^7 = 1/10,000,000\).

After each failed authentication attempt in the FIPS Approved mode, the NSS cryptographic module inserts a one-second delay before returning to the caller, allowing at most 60 authentication attempts during a one-minute period. Therefore, the probability of a successful random guess of the password during a one-minute period is less than \(60 * 1/10,000,000 = 0.6 * (1/100,000)\).

**Multiple Concurrent Operators**

The NSS cryptographic module doesn't allow concurrent operators.

- On a multi-user operating system, this is enforced by making the NSS certificate and private key databases readable and writable by the owner of the files only.

---

**Access Control Policy**

This section identifies the cryptographic keys and CSPs that the user has access to while performing a service, and the type of access the user has to the CSPs.

**Security-Relevant Information**

The NSS cryptographic module employs the following cryptographic keys and CSPs in the FIPS Approved mode of operation. Note that the private key database (key3.db/key4.db) mentioned below is outside the cryptographic boundary.

- **AES secret keys**: The module supports 128-bit, 192-bit, and 256-bit AES keys. The keys may be stored in memory or in the private key database (key3.db/key4.db).
- **Hash_DRBG**: Hash DRBG entropy - 880-bit value externally-obtained for module DRBG; stored in plaintext in volatile memory. Hash DRBG V value Internal Hash DRBG state value; stored in plaintext in volatile memory. Hash DRBG C value - Internal Hash DRBG state value; stored in plaintext in volatile memory.
- **Triple-DES secret keys**: 168-bit. The keys may be stored in memory or in the private key database (key3.db/key4.db).
- **HMAC secret keys**: HMAC key size must be greater than or equal to half the size of the hash function output. The keys may be stored in memory or in the private key database (key3.db/key4.db).
• DSA public keys and private keys: The module supports DSA key sizes of 512 and 1024 bits. Only DSA keys of 1024 bits may be used in the FIPS Approved mode of operation. The keys may be stored in memory or in the private key database (key3.db/key4.db).

• RSA public keys and private keys (used for digital signatures and key transport): The module supports RSA key sizes of 1024-8192 bits. The keys may be stored in memory or in the private key database (key3.db/key4.db).

• Diffie-Hellman primes: The module supports Diffie-Hellman prime sizes of 1024-2048 bits. The keys may be stored in memory or in the private key database (key3.db/key4.db).

• TLS premaster secret (used in deriving the TLS master secret): 48-byte. Stored in memory.

• TLS master secret (a secret shared between the peers in TLS connections, used in the generation of symmetric cipher keys, IVs, and MAC secrets for TLS): 48-byte. Stored in memory.

• Authentication data (passwords): Stored in the private key database (key3.db/key4.db).

---

**Self-Tests**

In the FIPS Approved mode of operation the cryptographic module does not allow critical errors to compromise security. Whenever a critical error (e.g., a self-test failure) is encountered, the cryptographic module enters an error state and the library needs to be reinitialized to resume normal operation. Reinitialization is accomplished by calling `FC_Finalize` followed by `FC_Initialize`.

Upon initialization of the cryptographic module library for the FIPS Approved mode of operation, the following power-up self-tests are performed:

a) Triple DES-ECB encrypt/decrypt
b) Triple DES-CBC encrypt/decrypt
c) AES-ECB encrypt/decrypt (128-bit, 192-bit, and 256-bit keys)
d) AES-CBC encrypt/decrypt (128-bit, 192-bit, and 256-bit keys)
e) SHA-1 hash
f) SHA-256 hash
g) SHA-384 hash
h) SHA-512 hash
i) HMAC-SHA-1/-SHA-256/-SHA-384/-SHA-512 keyed hash (296-bit key)
j) RSA encrypt/decrypt (1024-bit modulus \( n \))
k) RSA-SHA-256/-SHA-384/-SHA-512 signature generation (2048-bit modulus \( n \))
l) RSA-SHA-256/-SHA-384/-SHA-512 signature verification (2048-bit modulus \( n \))
m) DSA key pair generation (1024-bit prime modulus \( p \))
n) DSA signature generation (1024-bit prime modulus \( p \))
o) DSA signature verification (1024-bit prime modulus \( p \))

p) Random number generation, and

q) Software/firmware integrity test (the authentication technique is DSA with 1024-bit prime modulus \( p \))

Shutting down and restarting the NSS cryptographic module with the `FC_Finalize` and `FC_Initialize` functions executes the same power-up self-tests detailed above when initializing the module library for the FIPS Approved mode. This allows a user to execute these power-up self-tests on demand as defined in Section 4.9.1 of FIPS 140-2.

In the FIPS Approved mode of operation, the cryptographic module performs a pair-wise consistency test upon each invocation of RSA, and DSA key pair generation as defined in Section 4.9.2 of FIPS 140-2.

In the FIPS Approved mode of operation, the cryptographic module performs a continuous random number generator test upon each invocation of the pseudorandom number generator as defined in Section 4.9.2 of FIPS 140-2.

**Random Number Generator**

The cryptographic module performs pseudorandom number generation using NIST SP 800-90 Hash Deterministic Random Bit Generators.

The cryptographic module initializes its pseudorandom number generator by obtaining at least 110 bytes of random data from the operating system. The data obtained contains at least 440 bits of entropy. Extra entropy input is added by invoking a noise generator. Both initialization and noise generation are specific to the platform on which it was implemented. The pseudorandom number generator is seeded with noise derived from the execution environment such that the noise is not predictable. The source of noise is considered to be outside the logical boundary of the cryptographic module.

A product using the cryptographic module should periodically reseed the module's pseudorandom number generator with unpredictable noise by calling `FC_SeedRandom`. After \( 2^{46} \) calls to the random number generator the cryptographic module obtains another 110 bytes of random data from the operating system to reseed the random number generator.

**Module Ports and Interfaces**

The NSS cryptographic module is a software cryptographic implementation. No hardware or firmware components are included. All input to the module is via function arguments; all output is returned to the caller either as return codes or as updated memory objects pointed to by some of the arguments. All keys, encrypted data, and control information are exchanged through calls to library functions (logical interfaces). The physical ports, physical covers, doors, or openings; manual controls; and physical status indicators of the NSS cryptographic module are those of the general purpose computer it runs on.
Physical Cryptographic Boundary

Figure 1  Physical Cryptographic Boundary

Logical Cryptographic Boundary

Figure 2  NSS PKCS #11 Interface
Logical Interfaces

The following four logical interfaces have been designed within the NSS cryptographic module.

1. Data input interface: function input arguments that specify plaintext data; ciphertext or signed data; cryptographic keys (plaintext or encrypted) and initialization vectors; and passwords that are to be input to and processed by the NSS cryptographic module.

2. Data output interface: function output arguments that receive plaintext data; ciphertext data and digital signatures; and cryptographic keys (plaintext or encrypted) and initialization vectors from the NSS cryptographic module.

3. Control input interface: function calls, or input arguments that specify commands and control data (e.g., algorithms, algorithm modes, or module settings) used to control the operation of the NSS cryptographic module.

4. Status output interface: function return codes, error codes, or output arguments that receive status information used to indicate the status of the NSS cryptographic module.

The NSS cryptographic module uses different function arguments for input and output to distinguish between data input, control input, data output, and status output, to disconnect the logical paths followed by data/control entering the module and data/status exiting the module. The NSS cryptographic module doesn't use the same buffer for input and output. After the NSS cryptographic module is done with an input buffer that holds security-related information, it always zeroizes the buffer so that if the memory is later reused as an output buffer, no sensitive information may be inadvertently leaked.

PKCS #11

The logical interfaces of the NSS cryptographic module consist of the PKCS #11 (Cryptoki) API. The API itself defines the cryptographic boundary, i.e., all access to the cryptographic module is through this API. The module has three PKCS #11 tokens: two tokens that implement the non-FIPS Approved mode of operation, and one token that implements the FIPS Approved mode of operation. The FIPS PKCS #11 token is designed specifically for FIPS 140-2, and allows applications using the NSS cryptographic module to operate in a strictly FIPS mode.

The functions in the PKCS #11 API are listed in the table in the Specification of Services section.

Inhibition of Data Output

All data output via the data output interface is inhibited when the NSS cryptographic module is in the Error state or performing self-tests.

- In Error State: The Boolean state variable sftk_fatalError tracks whether the NSS cryptographic module is in the Error state. Most PKCS #11 functions, including all the functions that output data via the data output interface, check the sftk_fatalError state variable and, if it is true, return the CKR_DEVICE_ERROR error code immediately. Only the functions that shut down and restart the module, reinitialize the module, or output status information can be invoked in the Error state. These functions are FC_GetFunctionList, FC_Initialize, FC_Finalize, FC_GetInfo, FC_GetSlotList, FC_GetSlotInfo, FC_GetTokenInfo, FC_InitToken, FC_CloseSession, FC_CloseAllSessions, and FC_WaitForSlotEvent.

- During Self-Tests: The NSS cryptographic module performs power-up self-tests in the FC_Initialize function. Since no other PKCS #11 function (except FC_GetFunctionList) may be called before FC_Initialize returns successfully, all data output via the data output interface is inhibited while FC_Initialize is performing the self-tests.
Disconnecting the Output Data Path From the Key Processes

The NSS cryptographic module doesn't return the function output arguments until key generation or key zeroization is finished. Therefore, the logical paths used by output data exiting the module are logically disconnected from the processes/threads performing key generation and key zeroization.

Specification of Services

Cryptographic module services consist of public services, which require no user authentication, and private services, which require user authentication. Public services do not require access to the secret and private keys and other critical security parameters (CSPs) associated with the user. 

**Note:** CSPs are security-related information (e.g., secret and private keys, and authentication data such as passwords) whose disclosure or modification can compromise the security of a cryptographic module. Message digesting services are public only when CSPs are not accessed. Services which access CSPs (e.g., FC_GenerateKey, FC_GenerateKeyPair) require authentication. Some services require the user to assume the Crypto Officer or NSS User role. In the table below, the role is specified for each service. If the Role column is blank, no role needs to be assumed for that service; such a service (e.g., random number generation and hashing) does not affect the security of the module because it does not require access to the secret and private keys and other CSPs associated with the user. The table lists each service as an API function and correlates role, service type, and type of access to the cryptographic keys and CSPs. Access types \( R \), \( W \), and \( Z \) stand for Read, Write, and Zeroize, respectively.

<table>
<thead>
<tr>
<th>Service Category</th>
<th>Role</th>
<th>Function Name</th>
<th>Description</th>
<th>Cryptographic Keys and CSPs Accessed</th>
<th>Access Type, RWZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIPS 140-2 specific</td>
<td></td>
<td>FC_GetFunctionList</td>
<td>returns the list of function pointers for the FIPS Approved mode of operation</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>Module Initialization</td>
<td>FC_InitToken</td>
<td>initializes or reinitializes a token</td>
<td>password and all keys</td>
<td>Z</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FC_InitPIN</td>
<td>initializes the user's password, i.e., sets the user's initial password</td>
<td>password</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>General purpose</td>
<td>FC_Initialize</td>
<td>initializes the module library for the FIPS Approved mode of operation. This function provides the power-up self-test</td>
<td>none</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Function Name</td>
<td>Description</td>
<td>Arguments</td>
<td>Requirements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------</td>
<td>--------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC_Finalize</td>
<td>finalizes (shuts down) the module library</td>
<td>all keys</td>
<td>Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC_GetInfo</td>
<td>obtains general information about the module library</td>
<td>none</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC_GetSlotList</td>
<td>obtains a list of slots in the system</td>
<td>none</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC_GetSlotInfo</td>
<td>obtains information about a particular slot</td>
<td>none</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC_GetTokenInfo</td>
<td>obtains information about the token. This function provides the Show Status service.</td>
<td>none</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC_WaitForSlotEvent</td>
<td>This function is not supported by the NSS cryptographic module.</td>
<td>none</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC_GetMechanismList</td>
<td>obtains a list of mechanisms (cryptographic algorithms) supported by a token</td>
<td>none</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC_GetMechanismInfo</td>
<td>obtains information about a particular mechanism</td>
<td>none</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC_SetPIN</td>
<td>changes the user's password</td>
<td>password</td>
<td>RW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC_OpenSession</td>
<td>opens a connection(&quot;session&quot;) between an application and a particular token</td>
<td>none</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC_CloseSession</td>
<td>closes a session</td>
<td>keys of the session</td>
<td>Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC_CloseAllSessions</td>
<td>closes all sessions with a token</td>
<td>all keys</td>
<td>Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC_GetSessionInfo</td>
<td>obtains information about the session. This function provides the</td>
<td>none</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Parameters</td>
<td>Access Levels</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------------------</td>
<td>---------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC_GetOperationState</td>
<td>saves the state of the cryptographic operation in a session. This function is only implemented for message digest operations.</td>
<td>none</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC_SetOperationState</td>
<td>restores the state of the cryptographic operation in a session. This function is only implemented for message digest operations.</td>
<td>none</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC_Login</td>
<td>logs into a token</td>
<td>password</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_Logout</td>
<td>logs out from a token</td>
<td>none</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Object management</td>
<td>NSS User</td>
<td>FC_CreateObject</td>
<td>creates an object</td>
<td>key</td>
<td>W</td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_CopyObject</td>
<td>creates a copy of an object</td>
<td>original key</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_DestroyObject</td>
<td>destroys an object</td>
<td>key</td>
<td>Z</td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_GetObjectSize</td>
<td>obtains the size of an object in bytes</td>
<td>key</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_GetAttributeValue</td>
<td>obtains an attribute value of an object</td>
<td>key</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_SetAttributeValue</td>
<td>modifies an attribute value of an object</td>
<td>key</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_FindObjectsInit</td>
<td>initializes an object search operation</td>
<td>none</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_FindObjects</td>
<td>continues an object search operation</td>
<td>keys matching the search criteria</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_FindObjectsFinal</td>
<td>finishes an object search operation</td>
<td>none</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Encryption and decryption</td>
<td>NSS User</td>
<td>FC_EncryptInit</td>
<td>initializes an encryption operation</td>
<td>encryption key</td>
<td>R</td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_Encrypt</td>
<td>encrypts single-part data</td>
<td>encryption key</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_EncryptUpdate</td>
<td>continues a multiple-part encryption key</td>
<td>encryption key</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>Function</td>
<td>Description</td>
<td>Key Type</td>
<td>Notes</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>----------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FC_EncryptFinal</td>
<td>finishes a multiple-part encryption operation</td>
<td>encryption key</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FC_DecryptInit</td>
<td>initializes a decryption operation</td>
<td>decryption key</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FC_Decrypt</td>
<td>decrypts single-part encrypted data</td>
<td>decryption key</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FC_DecryptUpdate</td>
<td>continues a multiple-part decryption operation</td>
<td>decryption key</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FC_DecryptFinal</td>
<td>finishes a multiple-part decryption operation</td>
<td>decryption key</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FC_DigestInit</td>
<td>initializes a message-digesting operation</td>
<td>none</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FC_Digest</td>
<td>digests single-part data</td>
<td>none</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FC_DigestUpdate</td>
<td>continues a multiple-part digesting operation</td>
<td>none</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FC_DigestKey</td>
<td>continues a multipart message-digesting operation by digesting the value of a secret key as part of the data already digested</td>
<td>key</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FC_DigestFinal</td>
<td>finishes a multiple-part digesting operation</td>
<td>none</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Signature and NSS</td>
<td>FC_SignInit</td>
<td>initializes a signature</td>
<td>signing/HMAC</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>verification</td>
<td>User</td>
<td>operation</td>
<td>key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
<td>-----------------------------------------------</td>
<td>--------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_Sign</td>
<td>signs single-part data</td>
<td>signing/HMAC key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_SignUpdate</td>
<td>continues a multiple-part signature operation</td>
<td>signing/HMAC key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_SignFinal</td>
<td>finishes a multiple-part signature operation</td>
<td>signing/HMAC key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_SignRecoverInit</td>
<td>initializes a signature operation, where the data can be recovered from the signature</td>
<td>RSA signing key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_SignRecover</td>
<td>signs single-part data, where the data can be recovered from the signature</td>
<td>RSA signing key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_VerifyInit</td>
<td>initializes a verification operation</td>
<td>verification/HMAC key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_Verify</td>
<td>verifies a signature on single-part data</td>
<td>verification/HMAC key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_VerifyUpdate</td>
<td>continues a multiple-part verification operation</td>
<td>verification/HMAC key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_VerifyFinal</td>
<td>finishes a multiple-part verification operation</td>
<td>verification/HMAC key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_VerifyRecoverInit</td>
<td>initializes a verification operation where the data is recovered from the signature</td>
<td>RSA verification key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>FC_VerifyRecover</td>
<td>verifies a signature on single-part data, where the data is recovered from the signature</td>
<td>RSA verification key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual-function cryptographic operations</td>
<td>NSS User</td>
<td>FC_DigestEncryptUpdate</td>
<td>continues a multiple-part digesting and encryption operation</td>
<td>encryption key</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NSS User</td>
<td>FC_DecryptDigestUpdate</td>
<td>continues a multiple-part decryption and digesting operation</td>
<td>decryption key</td>
<td></td>
</tr>
<tr>
<td>NSS User</td>
<td>Function</td>
<td>Description</td>
<td>Key/Identifier</td>
<td>Parallelism</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FC_SignEncryptUpdate</td>
<td><strong>continues a multiple-part signing and encryption operation</strong></td>
<td>signing/HMAC key R</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>encryption key R</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FC_DecryptVerifyUpdate</td>
<td><strong>continues a multiple-part decryption and verify operation</strong></td>
<td>decryption key R</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>verification/HMAC key R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key</td>
<td>FC_GenerateKey</td>
<td><strong>generates a secret key</strong> (used by TLS to generate premaster secrets)</td>
<td>key W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>management</td>
<td>FC_GenerateKeyPair</td>
<td><strong>generates a public/private key pair</strong>. This function performs the pairwise consistency tests.</td>
<td>key pair W</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FC_WrapKey</td>
<td><strong>wraps (encrypts) a key</strong></td>
<td>wrapping key R</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>key to be wrapped R</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FC_UnwrapKey</td>
<td><strong>unwraps (decrypts) a key</strong></td>
<td>unwrapping key R</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>unwrapped key W</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FC_DeriveKey</td>
<td><strong>derives a key from base key</strong> (used by TLS to derive keys from the master secret)</td>
<td>base key R</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>derived key W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random</td>
<td>FC_SeedRandom</td>
<td><strong>mixes in additional seed material to the random number generator</strong></td>
<td>none RW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>number</td>
<td>FC_GenerateRandom</td>
<td><strong>generates random data</strong>. This function performs the continuous random number generator test.</td>
<td>none RW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>generation</td>
<td>FC_GetFunctionStatus</td>
<td><strong>a legacy function, which simply returns the value 0x00000051 (function not parallel)</strong></td>
<td>none -</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FC_CancelFunction</td>
<td><strong>a legacy function, which simply returns</strong></td>
<td>none -</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Note: The message digesting functions (except FC_DigestKey) do not require the user to assume an authorized role because they do not use any keys. FC_DigestKey computes the message digest (hash) of the value of a secret key. Therefore the user needs to assume the NSS User role for this service.

**Mitigation of Other Attacks**

The NSS cryptographic module is designed to mitigate the following attacks.

<table>
<thead>
<tr>
<th>Other Attacks</th>
<th>Mitigation Mechanism</th>
<th>Specific Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timing attacks on RSA</td>
<td><strong>RSA Blinding</strong></td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Timing attack on RSA was first demonstrated by Paul Kocher in 1996 [2], who contributed the mitigation code to our module. Most recently Boneh and Brumley [3] showed that RSA blinding is an effective defense against timing attacks on RSA.</td>
<td></td>
</tr>
<tr>
<td>Cache-timing attacks on the modular exponentiation operation used in RSA and DSA</td>
<td><strong>Cache Invariant Modular Exponentiation</strong></td>
<td>This mechanism requires intimate knowledge of the cache line sizes of the processor. The mechanism may be ineffective when the module is running on a processor whose cache line sizes are unknown.</td>
</tr>
<tr>
<td></td>
<td>This is a variant of a modular exponentiation implementation that Colin Percival [4] showed to defend against cache-timing attacks.</td>
<td></td>
</tr>
<tr>
<td>Arithmetic errors in RSA signatures</td>
<td><strong>Double-checking RSA Signatures</strong></td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Arithmetic errors in RSA signatures might leak the private key. Ferguson and Schneier [5] recommend that every RSA signature generation should verify the signature just generated.</td>
<td></td>
</tr>
</tbody>
</table>

**Access to Audit Data**

The NSS cryptographic module may use the *syslog* function and the audit mechanism provided by the operating system to audit events. Auditing is turned off by default. To turn on the auditing capability, you need to set the environment variable **NSS_ENABLE_AUDIT** to 1. You also need to configure the operating system's audit mechanism.

Access to the audit data is described in the next two subsections.
Access to Syslog Log Files

The NSS cryptographic module uses the syslog function to audit events, so the audit data are stored in the system log. Only the root user can modify the system log. On some platforms, only the root user can read the system log; on other platforms, all users can read the system log.

The system log is usually under the /var/adm or /var/log directory. The exact location of the system log is specified in the /etc/syslog.conf file. The NSS cryptographic module uses the default user facility and the info, warning, and err severity levels for its log messages. We give an example below.

The /etc/syslog.conf file has:

```
*.info;mail.none;authpriv.none;cron.none /var/log/messages
```

which specifies that /var/log/messages is the system log.

Sample Cryptographic Module Initialization Code

The following sample code uses NSPR functions (declared in the header file "prlink.h") for dynamic library loading and function symbol lookup.

```
#include "prlink.h"
#include "cryptoki.h"
#include <assert.h>
#include <stdio.h>
#include <string.h>

/*
* An extension of the CK_C_INITIALIZE_ARGS structure for the
* NSS cryptographic module. The 'LibraryParameters' field is
* used to pass instance-specific information to the library
* (like where to find its config files, etc).
*/
typedef struct CK_C_INITIALIZE_ARGS_NSS {
    CK_CREATEMUTEX CreateMutex;
    CK_DESTROYMUTEX DestroyMutex;
    CK_LOCKMUTEX LockMutex;
    CK_UNLOCKMUTEX UnlockMutex;
    CK_FLAGS flags;
    CK_CHAR_PTR *LibraryParameters;
    CK_VOID_PTR pReserved;
} CK_C_INITIALIZE_ARGS_NSS;

int main()
{
    char *libname;
    PRLibrary *lib;
    CK_C_GetFunctionList pFC_GetFunctionList;
    CK_FUNCTION_LIST_PTR pFunctionList;
    CK_RV rv;
    CK_C_INITIALIZE_ARGS_NSS initArgs;
    CK_SLOT_ID slotList[2], slotID;
```
CK_USLONG ulSlotCount;
CK_TOKEN_INFO tokenInfo;
CK_SESSION_HANDLE hSession;
CK_UTF8CHAR password[] = "1Mozilla";
PRStatus status;

/*
 * Get the platform-dependent library name of the NSS cryptographic module.
 */
libname = PR_GetLibraryName(NULL, "softokn3");
assert(libname != NULL);
lib = PR_LoadLibrary(libname);
assert(lib != NULL);
PR_FreeLibraryName(libname);

pFC_GetFunctionList =
    (CK_C_GetFunctionList)PR_FindFunctionSymbol(lib, "FC_GetFunctionList");
assert(pFC_GetFunctionList != NULL);
rv = (*pFC_GetFunctionList)(&pFunctionList);
assert(rv == CKR_OK);

/* Call FC_xxx via the function pointer pFunctionList->C_xxx */

initArgs.CreateMutex = NULL;
initArgs.DestroyMutex = NULL;
initArgs.LockMutex = NULL;
initArgs.UnlockMutex = NULL;
initArgs.flags = CKF_OS_LOCKING_OK;
initArgs.LibraryParameters = (CK_CHAR_PTR *)
    "configdir='.' certPrefix='' keyPrefix='' "
    "secmod='secmod.db' flags=";
initArgs.pReserved = NULL;
rv = pFunctionList->C_Initialize(&initArgs);
assert(rv == CKR_OK);

ulSlotCount = sizeof(slotList)/sizeof(slotList[0]);
rv = pFunctionList->C_GetSlotList(CK_TRUE, slotList, &ulSlotCount);
assert(rv == CKR_OK);
slotID = slotList[0];

rv = pFunctionList->C_OpenSession(slotID, CKF_RW_SESSION |
    CKF_SERIAL_SESSION, NULL, NULL, &hSession);
assert(rv == CKR_OK);

/* set the operator's initial password, if necessary */

rv = pFunctionList->C_GetTokenInfo(slotID, &tokenInfo);
assert(rv == CKR_OK);

if (!(tokenInfo.flags & CKF_USER_PIN_INITIALIZED)) {
    /*
     * As a formality required by the PKCS #11 standard, the
     * operator must log in as the PKCS #11 Security Officer (SO),
     * with the predefined empty string password, to set the
     * operator's initial password.
     */
    rv = pFunctionList->C_Login(hSession, CKU_SO, NULL, 0);
    assert(rv == CKR_OK);

    rv = pFunctionList->C_InitPIN(hSession,
                                   password, strlen(password));
    assert(rv == CKR_OK);

    /* log out as the PKCS #11 SO */
    rv = pFunctionList->C_Logout(hSession);
    assert(rv == CKR_OK);
}

/* the module is now ready for use */

/* authenticate the operator using a password */

rv = pFunctionList->C_Login(hSession, CKU_USER,
                           password, strlen(password));
assert(rv == CKR_OK);

/* use the module's services ... */

rv = pFunctionList->C_CloseSession(hSession);
assert(rv == CKR_OK);
rv = pFunctionList->C_Finalize(NULL);
assert(rv == CKR_OK);

status = PR_UnloadLibrary(lib);
assert(status == PR_SUCCESS); return 0;
}

The mode of operation of the NSS cryptographic module is determined by the second argument passed to the PR_FindFunctionSymbol function.

- For the non-FIPS Approved mode of operation, look up the standard PKCS #11 function C_GetFunctionList.
- For the FIPS Approved mode of operation, look up the alternative function FC_GetFunctionList.

**Obtaining Documentation, Support, and Security Guidelines**

For information on obtaining documentation, obtaining support, providing documentation feedback, security guidelines, and also recommended aliases and general Cisco documents, see the monthly What’s New in Cisco Product Documentation, which also lists all new and revised Cisco technical documentation at:
