Red Hat Enterprise Linux 6.6 NSS Module v3.14.3-22

FIPS 140-2 Security Policy

version 0.7

Last Update: 2016-02-12

Table of Contents

1. Cryptographic Module Specification

This document is the non-proprietary security policy for the Red Hat Enterprise Linux 6.6 NSS Module, and was prepared as part of the requirements for conformance to Federal Information Processing Standard (FIPS) 140-2, Level 2.

1.1. Description of the Module

The Red Hat Enterprise Linux 6.6 NSS Module (hereafter referred to as the “Module”) version 3.14.3-22 is a software library supporting FIPS 140-2 approved cryptographic algorithms. For the purposes of the FIPS 140-2 validation, its embodiment type is defined as multi-chip standalone. 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. It combines a vertical stack of Linux components intended to limit the external interface each separate component may provide.

The Module is FIPS 140-2 validated at overall Security Level 2 with levels for individual sections shown in the table below:

Security Component	FIPS 140-2 Security Level
Cryptographic Module Specification	2
Cryptographic Module Ports and Interfaces	2
Roles, Services and Authentication	2
Finite State Model	2
Physical Security	N/A
Operational Environment	2
Cryptographic Key Management	2
EMI/EMC	2
Self-Tests	2
Design Assurance	2
Mitigation of Other Attacks	2

Table 1: Security Level of the Module

The Module has been tested on the following platforms:

Manufacturer	Model	O/S & Ver.
HP (Hewlett-Packard)	ProLiant DL380p Gen8	Red Hat Enterprise Linux 6.6
IBM (International Business Machines)	System x3500 M4	Red Hat Enterprise Linux 6.6

Table 2: Tested Platforms

On each of the tested platforms listed above, the Module has been tested for the following configurations:

- 32-bit x86_64 with and without AES-NI enabled
- 64-bit x86_64 with and without AES-NI enabled

1.2. Description of the Approved Modes

The Module supports the following FIPS 140-2 approved algorithms:

Algorithm	Validation Certificate	Usage	Keys/CSPs
AES • CBC • ECB • CTR • GCM	Certs. 3076, 3077, 3078, 3079, 3080, 3081, 3082, 3083, 3084, 3085, 3086, 3087	encrypt/decrypt	AES keys 128, 192, 256 bits
Triple-DES • CBC • ECB	Certs. 1776, 1777, 1778, 1779	encrypt/decrypt	Triple-DES keys 168 bits
FIPS 186-4 DSA2	Certs. 892, 893, 894, 895	sign, verify, key generation, domain parameters generation and verification	DSA keys L=2048, N=224 L=2048, N=256 L=3072, N=256
FIPS 186-4 DSA2	Certs. 892, 893, 894, 895	verify	DSA keys L=1024, N=160
FIPS 186-4 RSA2 (PKCS #1.5)	Certs. 1577, 1578, 1579, 1580	sign, verify, and key generation	RSA keys 2048, 3072 bits
FIPS 186-4 RSA2 (PKCS #1.5)	Certs. 1577, 1578, 1579, 1580	verify	RSA keys 1024 bits
ECDSA	Certs. 554, 555, 556, 557	sign, verify, and key generation	ECDSA keys based on P-256, P-384, or P-521 curve
SP 800-90A Hash DRBG SHA-256	Certs. 603, 604, 605, 606	random number generation	Entropy input string, V and C
SHA-1 SHA-224 SHA-256 SHA-384 SHA-512	Certs. 2549, 2550, 2551, 2552	hashing	N/A
HMAC-SHA-1 HMAC-SHA224 HMAC-SHA256 HMAC-SHA384 HMAC-SHA512	Certs. 1933, 1934, 1935, 1936	message integrity	At least 112 bits HMAC Key
PRF as used in • TLSv1.0 • TLSv1.1 • TLSv1.2	CVL Certs. 368, 369, 370, 371	Key derivation	Pre-master secret

Table 3: Approved Algorithms

The Module supports AES-NI in 64-bit mode as the default option. In addition, C implementation of AES is available as a fall back option when the AES-NI capable CPU does not present in the Operational Environment.

The Module supports the following FIPS 140-2 non-approved but allowed algorithms:

Algorithm	Validation Certificate	Usage	Keys/CSPs
AES key wrapping (128, 192, 256 bits)	N/A	Key wrapping according to SP800-38F	AES keys
AES key wrapping (128, 192, 256 bits)	N/A	AES key wrapping using AES Encryption	AES keys
Triple-DES key wrapping (168 bits)	N/A	Triple-DES key wrapping using Triple-DES encryption	Triple-DES keys
RSA (encrypt, decrypt) with key size equal or larger than 2048 bits	N/A	Key wrapping	RSA keys
Diffie-Hellman with domain parameters larger or equal to 2048 bits	N/A	Key agreement	Diffie-Hellman private keys
EC Diffie-Hellman with curves P-256, P-384, P-521	N/A	Key agreement	EC Diffie-Hellman private keys

Table 4: Non-approved but allowed Algorithms

According to Table 2: Comparable strengths in NISP SP 800-57 Part1 (dated on March 8, 2007), the key sizes of RSA, Diffie-Hellman and EC Diffie-Hellman provides the following security strength for the corresponding key establishment method shown below:

1. *AES (key wrapping; key establishment methodology provides between 128 and 256 bits of encryption strength)*
2. *Triple-DES (key wrapping; key establishment methodology provides 112 bits of encryption strength)*
3. *RSA (key wrapping; key establishment methodology provides between 112 and 256 bits of encryption strength; non-compliant less than 112 bits of encryption strength)*
4. *Diffie-Hellman (key agreement; key establishment methodology provides between 112 and 256 bits of encryption strength; non-compliant less than 112 bits of encryption strength)*
5. *EC Diffie-Hellman (key agreement; key establishment methodology provides between 128 and 256 bits of encryption strength)*

However, the size alone does not determine the security strength of the RSA, Diffie-Hellman and EC Diffie-Hellman keys. Since the seed source for key generation is outside the logical boundary of the module, the following caveat is applicable:

The module generates cryptographic keys whose strengths are modified by available entropy.

The Module supports the following non-FIPS 140-2 approved algorithms:

Algorithm	Usage	Keys/CSPs
MD5	message digest	N/A
MD2	message digest	N/A
RC2	Encrypt/decrypt	RC2 key
Camellia	Encrypt/decrypt	Camellia key
J-PAKE	Key exchange	
DES	Encrypt/Decrypt	DES key
SEED	Encrypt/Decrypt	SEED key
CTS block chaining mode	Block chaining mode	N/A
DSA with key sizes not listed in Table 3	sign, verify, and key generation	DSA keys
RSA with key sizes not listed in Table 3	sign, verify, and key generation	RSA keys
Diffie-Hellman with domain parameters smaller than 2048 bits	Key agreement	Diffie-Hellman private keys
RSA (encrypt, decrypt) with key size smaller than 2048 bits	Key wrapping	RSA keys

Table 5. Non-Approved Algorithms

1.3. Cryptographic Boundary

The Module's physical boundary is the surface of the case of the platform (depicted in the hardware block diagram).

The Module's logical cryptographic boundary consists of the shared library files and their integrity check signature files, which are delivered through Red Hat Package Manager (RPM) as listed below:

- NSS softoken RPM file with version 3.14.3-22.el6, which contains the following files:
  - /usr/lib64/libnssdbm3.chk (64 bits)
  - /usr/lib64/libnssdbm3.so (64 bits)
  - /usr/lib64/libsoftokn3.chk (64 bits)
  - /usr/lib64/libsoftokn3.so (64 bits)
  - /usr/lib/libnssdbm3.chk (32 bits)
  - /usr/lib/libnssdbm3.so (32 bits)
  - /usr/lib/libsoftokn3.chk (32 bits)
  - /usr/lib/libsoftokn3.so (32 bits)
- NSS freebl RPM file with version 3.14.3-22.el6, which contains the following files:
  - /lib64/libfreeblpriv3.chk (64 bits)

- /lib64/libfreeblpriv3.so (64 bits)
- /lib/libfreeblpriv3.chk (32 bits)
- /lib/libfreeblpriv3.so (32 bits)

The module shall be installed and instantiated by the dracut-fips package with the RPM file version 004-356.el6_6.1. The dracut-fips RPM package is only used for the installation and instantiation of the Module. This code is not active when the Module is operational and does not provide any services to users interacting with the Module. Therefore the dracut-fips package is outside the Module's logical boundary.

1.3.1. Hardware Block Diagram

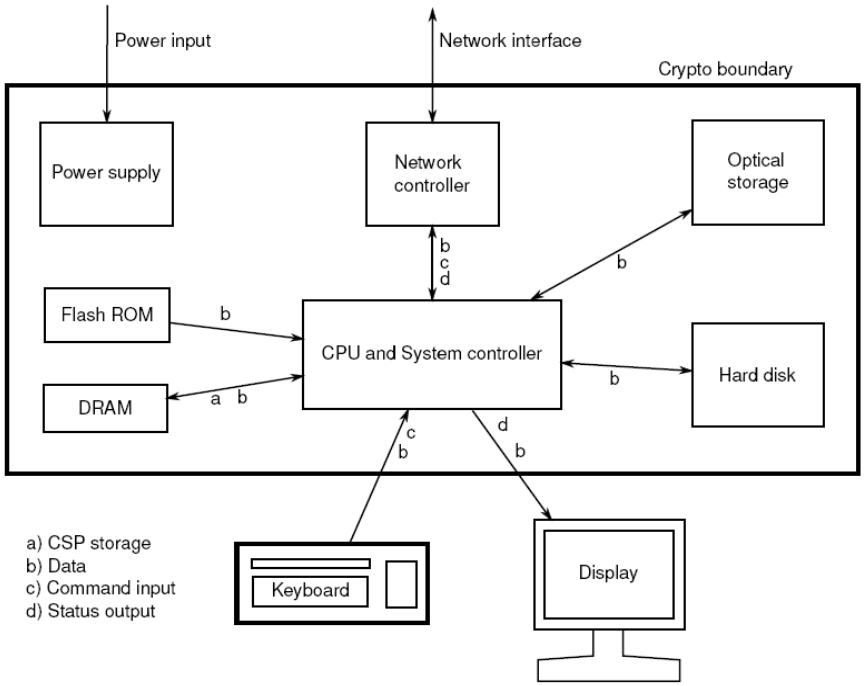

Figure 1. Hardware Block Diagram

1.3.2. Software Block Diagram

The NSS cryptographic module implements the PKCS #11 (Cryptoki) API. The API itself defines the logical cryptographic boundary, thus all implementation is inside the boundary. The diagram below shows the relationship of the layers.

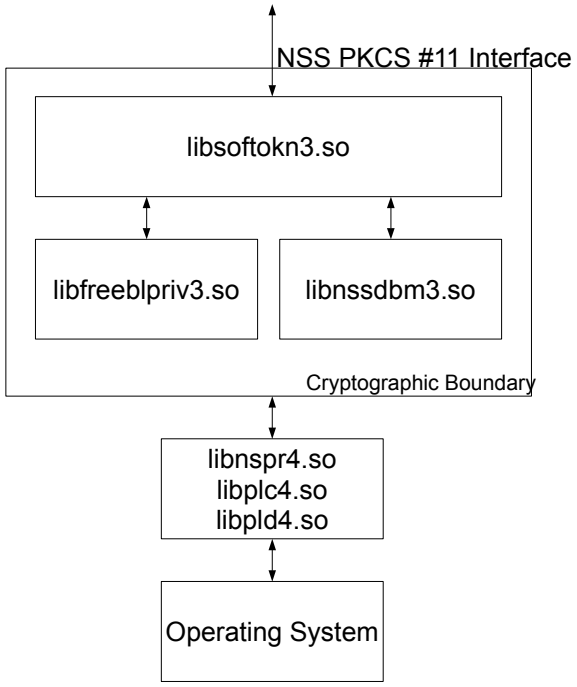

Figure 2. Software Block Diagram

2. Cryptographic Module Ports and Interfaces

The physical ports of the Module are the same as the computer system on which it executes. The logical interface is a C-language Application Program Interface (API) following the PKCS #11 API.

The Data Input interface consists of the input parameters of the API functions. The Data Output interface consists of the output parameters of the API functions. The Control Input interface consists of the actual API functions. The Status Output interface includes the return values of the API functions. The ports and interfaces are shown in the following table.

FIPS Interface	Physical Port	Module Interface
Data Input	N/A	API input parameters, kernel I/O – files on filesystem
Data Output	N/A	API output parameters, kernel I/O – files on filesystem
Control Input	N/A	API function calls, kernel I/O (/proc/sys/crypto/fips_enabled)
Status Output	N/A	API function calls
Power Input	PC Power Supply Port	N/A

Table 6: Ports and Interfaces

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.

2.1. 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 section about Specification of Services.

2.2. 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.

2.3. Disconnecting the Output Data Path from the Key Processes

During key generation and key zeroization, the NSS cryptographic module may perform audit logging, but the audit records do not contain sensitive information. The NSS cryptographic module does not 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.

3. Roles, Services and Authentication

This section defines the roles, services, and authentication mechanisms and methods with respect to the applicable FIPS 140-2 requirements.

3.1. Roles

There are two users of the Module:

- The NSS User role has access to all cryptographically secure services of the module (those that use the secret and private keys of the module) 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. Also, the crypto officer role can access other general-purpose services (such as message digest and random number generation services) and status services of the module. The Crypto-Officer does not have access to any service that utilizes the secret or private keys 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.

3.2. Role Assumption

The NSS cryptographic module implements a NSS User role and a Crypto-Officer role. The Crypto-Officer role is implicitly assumed by an operator while installing the module by following the instructions in the “Security Rules” section of this document and while performing the Crypto-Officer services on the module.

The module also implements a password-based authentication for the NSS User role. To perform sensitive NSS User role services using the cryptographic module, an operator must log into the module and perform an authentication procedure using the password information unique to the NSS User role operator. When passwords are passed via the API functions as input arguments, there is no visible display of the passwords, and the only feedback mechanism is the function return value. The function return value does not provide information that could be used to guess or determine the user's password. The password is initialized by the Crypto-Officer role as part of module initialization and can be changed by the NSS User role operator.

If a User-role service is called before the operator is authenticated, it returns the `CKR_USER_NOT_LOGGED_IN` error code. The operator must call the `FC_Login` function to provide the required authentication.

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.

3.3. 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 extended ASCII characters (accents such as 'é', 'è', special letters such as 'ß', 'å', or special characters such as '¿'). 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: the smallest class is the digit class (0-9) which size is 10. The probability for guessing every character is smaller than the probability of guessing the character if it was a digit, which is 1/10

Therefore, 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, which is smaller than the required threshold 1/1,000,000.

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).

3.4. 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.

Note: FIPS 140-2 Implementation Guidance Section 6.1 clarifies the use of a cryptographic module on a server.

When a cryptographic module is implemented in a server environment, the server application is the user of the cryptographic module. The server application makes the calls to the cryptographic module. Therefore, the server application is the single user of the cryptographic module, even when the server application is serving multiple clients.

3.5. Services

3.5.1. Calling Convention of 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. If an application initialized and uses both the FIPS Approved and non-FIPS Approved interfaces, each interface will contain it's own CSP which are not shared by the other interface. For an application to be in FIPS mode it must have only the FIPS Approved interfaces open. 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 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.

The following convention is used to describe API function calls. Again, FC_Initialize and NSC_Initialize are used as examples:

- When mentioned “call FC_Initialize,” the technical equivalent of “call the FC_Initialize function via the C_Initialize function pointer in the CK_FUNCTION_LIST structure returned by FC_GetFunctionList” is implied.
- When mentioned “call NSC_Initialize,” the technical equivalent of “call the NSC_Initialize function via the C_Initialize function pointer in the CK_FUNCTION_LIST structure returned by NSC_GetFunctionList” is implied.

3.5.2. API Functions

Cryptographic module services consists of Crypto-Officer services, which require no operator authentication, and User services, which require operator authentication. Crypto-Officer 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 available to Crypto-Officer only when CSPs are not accessed. Services which access CSPs (e.g., FC_GenerateKey, FC_GenerateKeyPair) require authentication. The table below 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.

Note: The message digesting functions (except FC_DigestKey) are allowed to the Crypto-Officer role and do not require NSS User role authentication to the module. These services do not use any keys of the module. FC_DigestKey computes the message digest (hash) of the value of a secret key, so it is available only to the NSS User role.

The list of algorithms supported by the Module is given in Table 3. For the column 'Role', 'U' corresponds to 'NSS User role' and 'CO' corresponds to 'Crypto Officer role'.

Service	Role	Function	Description	CSPs	Acc
FIPS 140-2 specific	CO	FC_GetFunctionList	returns the list of function pointers for the FIPS Approved mode of operation	none	-

Service	Role	Function	Description	CSPs	Acc
Module initialization	CO	FC_InitToken	initializes or reinitializes a token	User password and all keys	Z
	CO	FC_InitPIN	initializes the user's password, i.e., sets the user's initial password	User password	W
General Purpose	CO	FC_Initialize	initializes the module library for the FIPS approved mode of operation. This function provides the power-up self-test service.	none	-
	CO	FC_Finalize	finalizes (shuts down) the module library	All keys	Z
	CO	FC_GetInfo	obtains general information about the module library	none	-
Slot and token management	CO	FC_GetSlotList	obtains a list of slots in the system	none	-
	CO	FC_GetSlotInfo	obtains information about a particular slot	none	-
	CO	FC_GetTokenInfo	obtains information about the token. This function provides the Show Status service.	none	-
	CO	FC_WaitForSlotEvent	This function is not supported by the NSS cryptographic module.	none	-
	CO	FC_GetMechanismList	obtains a list of mechanisms (cryptographic algorithms) supported by a token	none	-
	CO	FC_GetMechanismInfo	obtains information about a particular mechanism	none	-

Service	Role	Function	Description	CSPs	Acc
	U	FC_SetPIN	changes the user's password	User password	RW
Session management	CO	FC_OpenSession	opens a connection ("session") between an application and a particular token	none	-
	CO	FC_CloseSession	closes a session	All keys for the session	Z
	CO	FC_CloseAllSessions	closes all sessions with a token	All keys	Z
	CO	FC_GetSessionInfo	obtains information about the session. This function provides the Show Status service.	none	-
	CO	FC_GetOperationState	saves the state of the cryptographic operation in a session. This function is only implemented for message digest operations.	none	-
	CO	FC_SetOperationState	restores the state of the cryptographic operation in a session. This function is only implemented for message digest operations.	none	-
	U	FC_Login	logs into a token	User password	R
	U	FC_Logout	logs out from a token	none	-
Object management	U	FC_CreateObject	creates an object	key	W
	U	FC_CopyObject	creates a copy of an object	Original key	R
				New key	W
	U	FC_DestroyObject	destroys an object	key	Z
	U	FC_GetObjectSize	obtains the size of an object in bytes	key	R
	U	FC_GetAttributeValue	obtains an attribute value of an object	key	R

Service	Role	Function	Description	CSPs	Acc
	U	FC_SetAttributeValue	modifies an attribute value of an object	key	W
	U	FC_FindObjectsInit	initializes an object search operation	none	-
	U	FC_FindObjects	continues an object search operation	Keys matching the search criteria	R
	U	FC_FindObjectsFinal	finishes an object search operation	none	-
Encryption and decryption	U	FC_EncryptInit	initializes an encryption operation	AES / Triple-DES encryption key	R
	U	FC_Encrypt	Encrypts single-part data	AES / Triple-DES encryption key	R
	U	FC_EncryptUpdate	continues a multiple-part encryption operation	AES / Triple-DES encryption key	R
	U	FC_EncryptFinal	finishes a multiple-part encryption operation	AES / Triple-DES encryption key	R
	U	FC_DecryptInit	initializes a decryption operation	AES / Triple-DES encryption key	R
	U	FC_Decrypt	decrypts single-part encrypted data	AES / Triple-DES encryption key	R
	U	FC_DecryptUpdate	continues a multiple-part decryption operation	AES / Triple-DES encryption key	R
	U	FC_DecryptFinal	finishes a multiple-part decryption operation	AES / Triple-DES encryption key	R
Message digest	CO	FC_DigestInit	initializes a message-digesting operation	none	-
	CO	FC_Digest	Digests single-part data	none	-
	CO	FC_DigestUpdate	continues a multiple-part digesting operation	none	-
	U	FC_DigestKey	continues a multi-part message-digesting operation by digesting the value of a secret key as part of the data	HMAC key	R

Service	Role	Function	Description	CSPs	Acc
			already digested		
	CO	FC_DigestFinal	finishes a multiple-part digesting operation	none	-
Signature generation and verification	U	FC_SignInit	initializes a signature operation	RSA / DSA / ECDSA signing key, HMAC key	R
	U	FC_Sign	signs single-part data	RSA / DSA / ECDSA signing key, HMAC key	R
	U	FC_SignUpdate	continues a multiple-part signature operation	RSA / DSA / ECDSA signing key, HMAC key	R
	U	FC_SignFinal	finishes a multiple-part signature operation	RSA / DSA / ECDSA signing key, HMAC key	R
	U	FC_SignRecoverInit	initializes a signature operation, where the data can be recovered from the signature	RSA / DSA / ECDSA signing key	R
	U	FC_SignRecover	signs single-part data, where the data can be recovered from the signature	RSA / DSA / ECDSA signing key	R
	U	FC_VerifyInit	initializes a verification operation	RSA / DSA / ECDSA verification key, HMAC key	R
	U	FC_Verify	verifies a signature on single-part data	RSA / DSA / ECDSA verification key, HMAC key	R
	U	FC_VerifyUpdate	continues a multiple-part verification operation	RSA / DSA / ECDSA verification key, HMAC key	R
	U	FC_VerifyFinal	finishes a multiple-part verification operation	RSA / DSA / ECDSA verification key, HMAC key	R
	U	FC_VerifyRecoverInit	initializes a verification operation where the data is recovered from the signature	RSA / DSA / ECDSA verification key	R
	U	FC_VerifyRecover	verifies a signature	RSA / DSA / ECDSA	R

Service	Role	Function	Description	CSPs	Acc
			on single-part data, where the data is recovered from the signature	verification key	
Dual-function cryptographic operations	U	FC_DigestEncryptUpdate	continues a multiple-part digesting and encryption operation	RSA / DSA / ECDSA encryption key	R
	U	FC_DecryptDigestUpdate	continues a multiple-part decryption and digesting operation	RSA / DSA / ECDSA decryption key	R
	U	FC_SignEncryptUpdate	continues a multiple-part signing and encryption operation	RSA / DSA / ECDSA signing key, HMAC key	R
				RSA / DSA / ECDSA encryption key	R
	U	FC_DecryptVerifyUpdate	continues a multiple-part decryption and verify operation	RSA / DSA / ECDSA verification key, HMAC key	R
				RSA / DSA / ECDSA decryption key	R
Key management	U	FC_GenerateKey	generates a secret key (used by TLS to generate premaster secrets)	AES keys, Triple-DES keys, TLS pre-master secret	W
	U	FC_GenerateKeyPair	generates a public/private key pair. This function performs the pair-wise consistency tests	RSA / DSA / ECDSA key pair	W
	U	FC_WrapKey	wraps (encrypts) a key using one of the following mechanisms allowed in FIPS mode through December 31st 2017 per IG D.9: (1) RSA encryption (2) AES Key Wrapping based on SP 800-38F (3) AES encryption (4) Triple-DES encryption	Wrapping key	R
				Key to be wrapped	R

Service	Role	Function	Description	CSPs	Acc
	U	FC_UnwrapKey	unwraps (decrypts) a key using one of the following mechanisms allowed in FIPS mode through December 31st 2017 per IG D.9: (1) RSA decryption (2) AES Key Wrapping based on SP 800-38F (3) AES decryption (4) Triple-DES decryption	Unwrapping key	R
				Uwrapped key	W
	U	FC_DeriveKey	derives a key from a master secret key (used by TLS to derive keys from the master secret)	Master secret key	R
				Derived key	W
Random number generation	CO	FC_SeedRandom	mixes in additional seed material to the random number generator	entropy string for seed, DRBG C and V internal state values	RW
	CO	FC_GenerateRandom	generates random data. This function performs the continuous random number generator test.	random data, DRBG C and V internal state values	RW
Parallel function management	CO	FC_GetFunctionStatus	a legacy function, which simply returns the value 0x00000051 (function not parallel)	none	-
	CO	FC_CancelFunction	a legacy function, which simply returns the value 0x00000051 (function not parallel)	none	-

Table 7: Service Details

NOTE:

1. *'Original key' and 'New key' are the secret keys or public private key pairs.*
2. *'Wrapping key' corresponds to the secret key or private key used to wrap another key*

3. *'Key to be wrapped' is the key that is wrapped by the 'wrapping key'*
4. *'Unwrapping key' corresponds to the secret key or public key used to unwrap another key*
5. *'Unwrapped key' is the plaintext key that has not been wrapped by a 'wrapping key'*
6. *'Derived key' is the key obtained by a key derivation function which takes the 'master secret key' as input*

The following list of services are available in non-FIPS mode. Note that they are the same as the FIPS mode services listed in the table above, but with the NSC_xxx tag in front of the API function, instead of FC_xxx. The behaviors of these functions are identical to their FIPS mode counterparts. If applicable, the non-approved algorithms used by the service is listed in the parenthesis.

- NSC_GetFunctionList (none)
- NSC_InitToken (none)
- NSC_InitPIN (none)
- NSC_Initialize (none)
- NSC_Finalize (none)
- NSC_GetInfo (none)
- NSC_GetSlotList (none)
- NSC_GetSlotInfo (none)
- NSC_GetTokenInfo (none)
- NSC_WaitForSlotEvent (none)
- NSC_GetMechanismList (none)
- NSC_GetMechanismInfo (none)
- NSC_SetPIN (none)
- NSC_OpenSession (none)
- NSC_CloseSession (none)
- NSC_CloseAllSessions (none)
- NSC_GetSessionInfo (none)
- NSC_GetOperationState (none)
- NSC_SetOperationState (none)
- NSC_Login (none)
- NSC_Logout (none)
- NSC_CreateObject (none)
- NSC_CopyObject (none)
- NSC_DestroyObject (none)
- NSC_GetObjectSize (none)
- NSC_GetAttributeValue (none)
- NSC_SetAttributeValue (none)

- NSC_FindObjectsInit (none)
- NSC_FindObjects (none)
- NSC_FindObjectsFinal (none)
- NSC_EncryptInit (RC2, Camellia, DES, SEED, CTS block chaining mode)
- NSC_Encrypt (RC2, Camellia, DES, SEED, CTS block chaining mode)
- NSC_EncryptUpdate (RC2, Camellia, DES, SEED, CTS block chaining mode)
- NSC_EncryptFinal (RC2, Camellia, DES, SEED, CTS block chaining mode)
- NSC_DecryptInit (RC2, Camellia, DES, SEED, CTS block chaining mode)
- NSC_Decrypt (RC2, Camellia, DES, SEED, CTS block chaining mode)
- NSC_DecryptUpdate (RC2, Camellia, DES, SEED, CTS block chaining mode)
- NSC_DecryptFinal (RC2, Camellia, DES, SEED, CTS block chaining mode)
- NSC_DigestInit (MD5, MD2)
- NSC_Digest (MD5, MD2)
- NSC_DigestUpdate (MD5, MD2)
- NSC_DigestKey (MD5, MD2)
- NSC_DigestFinal (MD5, MD2)
- NSC_SignInit (RSA and DSA with key sizes not listed in Table 3)
- NSC_Sign (RSA and DSA with key sizes not listed in Table 3)
- NSC_SignUpdate (RSA and DSA with key sizes not listed in Table 3)
- NSC_SignFinal (RSA and DSA with key sizes not listed in Table 3)
- NSC_SignRecoverInit (RSA and DSA with key sizes not listed in Table 3)
- NSC_SignRecover (RSA and DSA with key sizes not listed in Table 3)
- NSC_VerifyInit (RSA and DSA with key sizes not listed in Table 3)
- NSC_Verify (RSA and DSA with key sizes not listed in Table 3)
- NSC_VerifyUpdate (RSA and DSA with key sizes not listed in Table 3)
- NSC_VerifyFinal (RSA and DSA with key sizes not listed in Table 3)
- NSC_VerifyRecoverInit (RSA and DSA with key sizes not listed in Table 3)
- NSC_VerifyRecover (RSA and DSA with key sizes not listed in Table 3)
- NSC_DigestEncryptUpdate (RSA and DSA with key sizes not listed in Table 3)
- NSC_DecryptDigestUpdate (RSA and DSA with key sizes not listed in Table 3)
- NSC_SignEncryptUpdate (RSA and DSA with key sizes not listed in Table 3)
- NSC_DecryptVerifyUpdate (RSA and DSA with key sizes not listed in Table 3)
- NSC_GenerateKey (none)
- NSC_GenerateKeyPair (none)
- NSC_WrapKey (CAMELLIA, SEED, DES, RC4, RC2, RSA Key wrapping with key size smaller than 2048 bits)

- NSC_UnwrapKey (CAMELLIA, SEED, DES, RC4, RC2, RSA Key wrapping with key size smaller than 2048 bits)
- NSC_DeriveKey (none)
- NSC_SeedRandom (none)
- NSC_GenerateRandom (none)
- NSC_GetFunctionStatus (none)
- NSC_CancelFunction (none)

4. Physical Security

The Module comprises of software only and thus does not claim any physical security.

5. Operational Environment

This Module operates in a modifiable Operational Environment per the FIPS 140-2 definition.

The underlying operating system of Red Hat Enterprise Linux 6 is evaluated according to Common Criteria at EAL4 – certification IDs of BSI-DSZ-CC-0924 as well as BSI-DSZ-CC-0754 claiming compliance to the OSPP.

5.1. Policy

The operating system is restricted to a single operator mode of operation (i.e., concurrent operators are explicitly excluded).

The application that makes calls to the Module is the single user of the Module, even when the application is serving multiple clients.

In FIPS approved mode, the ptrace(2) system call, the debugger (gdb(1)), and strace(1) shall not be used. In addition, other tracing mechanisms offered by the Linux environment, such as ftrace or systemtap, shall not be used.

6. Cryptographic Key Management

The application that uses the Module is responsible for appropriate destruction and zeroization of the key material. The library provides functions for key allocation and destruction, which overwrites the memory that is occupied by the key information with “zeros” before it is deallocated.

The following table provides a summary of the Keys/CSPs in the module:

Keys/CSPs	Keys/CSPs length	Key Generation	Key Storage	Key Entry/Output	Key Zeroization
AES Keys	128, 192 or 256 bits	Use of the module's DRBG	Application memory or key database	Encrypted through key wrapping using FC_WrapKey	Automatic zeroized when freeing the cipher handle
Triple-DES Keys	168 bits	Use of the module's DRBG	Application memory or key database	Encrypted through key wrapping using FC_WrapKey	Automatic zeroized when freeing the cipher handle
DSA private keys	2048 and 3072 bits	Use of the module's DRBG and the modules DSA key generation mechanism	Application memory or key database	Encrypted through key wrapping using FC_WrapKey	Automatic zeroized when freeing the cipher handle
RSA private keys	2048 and 3072 bits	Use of the module's DRBG and the modules RSA key generation mechanism	Application memory or key database	Encrypted through key wrapping using FC_WrapKey	Automatic zeroized when freeing the cipher handle
ECDSA private keys	EC key lengths according to NIST curves P-256, P-384 and P-521	Use of the module's DRBG and the modules ECDSA key generation mechanism	Application memory or key database	Encrypted through key wrapping using FC_WrapKey	Automatic zeroized when freeing the cipher handle
DRBG Entropy string	880 bits (twice the required size of at least 440 bits defined in SP800-90A)	The seed data obtained from hardware random number generator /dev/urandom	Application memory	N/A	Automatic zeroized when freeing DRBG handle
DRBG C, V values	880 bits	Based on entropy string as defined in SP800-90A	Application memory	N/A	Automatic zeroized when freeing DRBG handle

HMAC Keys	At least 112 bits	Use of the module's DRBG	Application memory or key data base	Encrypted through key wrapping using FC_WrapKey	Automatic zeroized when freeing the cipher handle
TLS pre-master secret	Pre-master secret length according to TLS v1.0, 1.1 and 1.2	Use of the module's DRBG together with Diffie-Hellman/EC Diffie-Hellman	Application memory	N/A	Automatic zeroized when freeing the cipher handle
TLS master secret	As required by TLS ciphersuite	Use of the TLS pre-master secret and PRF	Application memory	N/A	Automatic zeroized when freeing the cipher handle
User Passwords	N/A	Input through keyboard by the Crypto Officer when initiating the module	Stored in salted form in the key database	Input through keyboard by the User for authentication	Zeroized when the module is re-initialized or overwritten when the user changes its password

Table 8: Keys/CSPs

6.1. Random Number Generation

The cryptographic module performs pseudorandom number generation using NIST SP 800-90 Hash Deterministic Random Bit Generator using SHA-256. The cryptographic module initializes its pseudorandom number generator by obtaining 880 bits of random data from the operating system via /dev/urandom. The seed length is determined by the underlying SHA algorithms based on the requirement listed in section 10.1 of SP 800-90A. The entropy source /dev/urandom provides at least 110 bytes of random data available to the cryptographic module to obtain.

Reseeding is performed by pulling more data from /dev/urandom. The module provides at least 112 bits of entropy.

A product using the cryptographic module should periodically reseed the module's pseudorandom number generator with unpredictable noise by calling FC_SeedRandom. After 2^{48} calls to the random number generator the cryptographic module obtains another 110 bytes of random data from the operating system via /dev/urandom.

The module generates keys whose strengths are modified by available entropy.

6.2. Key/CSP Access And Storage

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.

The NSS cryptographic module employs the cryptographic keys and CSPs in the FIPS Approved mode of operation listed in the aforementioned table. Note that the private key database (provided with the files key3.db/key4.db) mentioned above is outside the cryptographic boundary.

Note: The NSS cryptographic module does not implement the TLS protocol. The NSS cryptographic

module implements the cryptographic operations, including TLS-specific key generation and derivation operations, that can be used to implement the TLS protocol.

Public and private keys are provided to the Module by the calling process, and are destroyed when released by the appropriate API function calls.

6.3. Key/CSP Zeroization

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 gets 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.

7. Electromagnetic Interference/Electromagnetic Compatibility (EMI/EMC)

Product Name and Model: HP ProLiant DL380p Gen8

Regulatory Model Number: HSTNS-5163

Product Options: All

EMC: Class A

Product Name and Model: IBM System x3500 M4

Regulatory Model Number: 7383-AC1

Product Options: All

EMC: Class A

8. Self-Tests

FIPS 140-2 requires that the Module perform self-tests to ensure the integrity of the Module and the correctness of the cryptographic functionality at start up. In addition, some functions require continuous verification of function, such as the Random Number Generator. All of these tests are listed and described in this section.

The Module performs both power-up self-tests (at module initialization) and continuous condition tests (during operation). No operator intervention is required during the running of the self-tests. Input, output, and cryptographic functions cannot be performed while the Module is in a self-test or error state because the Module is globally in self-test mode, and will not return to the calling application until the power-up self-tests are completed successfully. If any self-tests fail, the Module enters the error state and subsequent calls to the Module will also fail - thus no further cryptographic operations are possible. The Module returns the error code CKR_DEVICE_ERROR to the calling application to indicate the error state.

The power-up self tests can be performed on demand by re-initializing the Module.

See section 9.3 for descriptions of possible self-test errors and recovery procedures.

8.1. Power-Up Tests

The following table provides the lists of Known-Answer Test (KAT) and Integrity Test as the power-up self-tests:

Algorithm	Test
AES	KAT: encryption and decryption are tested separately
Triple-DES	KAT: encryption and decryption are tested separately
DSA	KAT: signature generation and verification are tested separately
RSA	KAT: encryption and decryption are tested separately KAT: signature generation and verification are tested separately
ECDSA	KAT: signature generation and verification are tested separately
SP 800-90A Hash DRBG	KAT
HMAC-SHA-1, -244, -256, -384, -512	KAT
SHA-1, -224, -256, -384, -512	KAT
Module integrity	DSA 2048 bits signature verification with SHA-256

Table 9: Module Self-Tests

8.2. Conditional Tests

The following table provides the lists of Pairwise Consistency Test (PCT) and Continuous Random Number Generation Test (CRNGT) as the conditional self-tests:

Algorithm	Test
DSA	PCT: signature generation and verification are tested separately
ECDSA	PCT: signature generation and verification are tested separately
RSA	PCT: signature generation and verification are tested separately, encryption and decryption are tested separately
DRBG SP800-90A	CRNGT

Table 10: Module Conditional Tests

9. Guidance

9.1. Crypto Officer Guidance

The version of the RPMs containing the FIPS validated Module is stated in section 1.1. The RPM packages forming the Module can be installed by standard tools recommended for the installation of RPM packages on a Red Hat Enterprise Linux system (for example, yum, rpm, and the RHN remote management tool). All RPM packages are signed with the Red Hat build key, which is an RSA 2048 bit key using SHA-256 signatures. The signature is automatically verified upon installation of the RPM package. If the signature cannot be validated, the RPM tool rejects the installation of the package. In such a case, the crypto officer is requested to obtain a new copy of the module RPMs from Red Hat.

In addition, to support the Module, the NSPR library must be installed that is offered by the underlying operating system.

Only the cipher types listed in section 1.2 are allowed to be used.

To bring the Module into FIPS approved mode, perform the following:

1. Install the dracut-fips package:
   ```
   # yum install dracut-fips
   ```
2. Recreate the INITRAMFS image:
   ```
   # dracut -f
   ```

After regenerating the initramfs, the Crypto Officer has to append the following string to the kernel command line by changing the setting in the boot loader:

```
fips=1
```

If /boot or /boot/efi resides on a separate partition, the kernel parameter boot=<partition of /boot or /boot/efi> must be supplied. The partition can be identified with the command

```
"df /boot"
```

or

```
"df /boot/efi"
```

respectively. For example:

```
$ df /boot
Filesystem         1K-blocks   Used        Available         Use%  Mounted on
/dev/sda1           233191     30454       190296            14%   /boot
```

The partition of /boot is located on /dev/sda1 in this example. Therefore, the following string needs to be appended to the kernel command line:

```
"boot=/dev/sda1"
```

Reboot to apply these settings.

If an application that uses the Module for its cryptography is put into a chroot environment, the Crypto Officer must ensure one of the above methods is available to the Module from within the chroot environment to ensure entry into FIPS approved mode. Failure to do so will not allow the application to properly enter FIPS approved mode.

The version of the RPM containing the validated Module is the version listed in chapter 1. The integrity of the RPM is automatically verified during the installation of the Module and the Crypto Officer shall not install the RPM file if the RPM tool indicates an integrity error.

9.1.1. Access to Audit Data

The Module may use the Unix syslog function and the audit mechanism provided by the operating system to audit events. Auditing is turned off by default. Auditing capability must be turned on as part of the initialization procedures by setting the environment variable NSS_ENABLE_AUDIT to 1. The Crypto-Officer must also configure the operating system's audit mechanism.

The 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/log directory. The exact location of the system log is specified in the /etc/syslog.conf file. The Module uses the default user facility and the info, warning, and err severity levels for its log messages.

The NSS cryptographic module can also be configured to use the audit mechanism provided by the operating system to audit events. The audit data would then be stored in the system audit log. Only the root user can read or modify the system audit log. To turn on this capability it is necessary to create a symbolic link from the library file /usr/lib/libaudit.so.0 to /usr/lib/libaudit.so.1.0.0 (on 32-bit platforms) and /usr/lib64/libaudit.so.0 to /usr/lib64/libaudit.so.1.0.0 (on 64-bit platforms).

9.2. User Guidance

The Module must be operated in FIPS approved mode to ensure that FIPS 140-2 validated cryptographic algorithms and security functions are used.

The following module initialization steps must be followed by the Crypto-Officer before starting to use the NSS module:

- Set the environment variable `NSS_ENABLE_AUDIT` to 1 before using the NSS module with an application.
- Use the application to get the function pointer list using the NSS API "`FC_GetFunctionList`".
- Use the API `FC_Initialize` to initialize the module. Using the `FC_GetFunctionList` above ensured that we selected FIPS mode, and the subsequent FC_Initialize call then initializes the module in FIPS-mode. Ensure that this returns CKR_OK. A return code other than CKR_OK will mean the FIPS-mode was not enabled, and in that case, the module must be reset and initialized again.
- For the first login, provide a NULL password and login using the function pointer `C_Login`, which will in-turn call `FC_Login` API of the module. This is required to set the initial NSS User password.
- Now, set the initial NSS User role password using the function pointer `C_InitPIN`. This will call the module's API `FC_InitPIN` API. Then, logout using the function pointer `C_Logout`, which will call the module's API `FC_Logout`.
- The NSS User role can now be assumed on the module by logging in using the User password. And the Crypto-Officer role can be implicitly assumed by performing the Crypto-Officer services as listed in Section "Specification of Services" of this document.

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 and all data stored in these databases is considered stored in plaintext. The interface code of the NSS cryptographic module that accesses data stored in the database is considered part of the cryptographic boundary.

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 must be passed to the calling application only in encrypted (wrapped) form with FC_WrapKey and entered from calling application only in encrypted (wrapped) form with FC_UnwrapKey. The cryptographic algorithms allowed for this purpose in FIPS-mode are AES, Triple-DES, RSA using the corresponding Approved modes and key sizes. 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.

Automated key transport methods must use FC_WrapKey and FC_UnwrapKey to input or output secret and private keys to or from the module.

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.

9.2.1. AES GCM Guidance

The AES GCM IV generation is compliant with IETF RFC 5288. The GCM block chaining mode shall only be used together with the TLS protocol. This protocol ensures that the encryption operation is invoked with updated IVs for each processed data packet.

The API for GCM encryption restricts the size of the processed data packet to 2^{32} bytes. Combining that with the requirement to use the GCM cipher with TLS only automatically implies that never more than 2^{28} AES blocks are encrypted with the same key and IV combination.

9.2.2. RSA and DSA Keys

The Module allows the use of 1024 bit RSA and DSA keys for legacy purposes, including signature generation.

As per SP800-131A, RSA and DSA must be used with either 2048 bit keys or 3072 bit keys. To comply with the requirements of FIPS 140-2, a user must therefore only use keys with 2048 bits or 3072 bits.

9.3. Handling Self-Test Errors

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.

10. Mitigation of Other Attacks

The Module is designed to mitigate the following attacks.

Attack	Mitigation Mechanism	Specific Limit
Timing attacks on RSA	**RSA blinding** Timing attack on RSA was first demonstrated by Paul Kocher in 1996 [16], who contributed the mitigation code to our module. Most recently Boneh and Brumley [17] showed that RSA blinding is an effective defense against timing attacks on RSA.	None
Cache-timing attacks on the modular exponentiation operation used in RSA and DSA	**Cache invariant modular exponentiation** This is a variant of a modular exponentiation implementation that Colin Percival [18] showed to defend against cache-timing attacks	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.
Arithmetic errors in RSA signatures	**Double-checking RSA signatures** Arithmetic errors in RSA signatures might leak the private key. Ferguson and Schneier [19] recommend that every RSA signature generation should verify the signature just generated.	None

11. Glossary and Abbreviations

AES	Advanced Encryption Specification
CAVP	Cryptographic Algorithm Validation Program
CBC	Cypher Block Chaining
CCM	Counter with Cipher Block Chaining-Message Authentication Code
CFB	Cypher Feedback
CMT	Cryptographic Module Testing
CMVP	Cryptographic Module Validation Program
CSP	Critical Security Parameter
CVT	Component Verification Testing
DES	Data Encryption Standard
DSA	Digital Signature Algorithm
ECB	Electronic Code Book
FSM	Finite State Model
HMAC	Hash Message Authentication Code
MAC	Message Authentication Code
NIST	National Institute of Science and Technology
NVLAP	National Voluntary Laboratory Accreditation Program
OFB	Output Feedback
O/S	Operating System
PRNG	Pseudo Random Number Generator
RNG	Random Number Generator
RSA	Rivest, Shamir, Addleman
SDK	Software Development Kit
SHA	Secure Hash Algorithm
SHS	Secure Hash Standard
SLA	Service Level Agreement
SOF	Strength of Function
SSH	Secure Shell
TDES	Triple DES
UI	User Interface

12. References

[1] OpenSSL man pages where crypto(3) provides the introduction and link to all OpenSSL APIs regarding the cryptographic operation and ssl(3) to all OpenSSL APIs regarding the SSL/TLS protocol family

[2] FIPS 140-2 Standard, http://csrc.nist.gov/groups/STM/cmvp/standards.html

[3] FIPS 140-2 Implementation Guidance, http://csrc.nist.gov/groups/STM/cmvp/standards.html

[4] FIPS 140-2 Derived Test Requirements,http://csrc.nist.gov/groups/STM/cmvp/standards.html

[5] FIPS 197 Advanced Encryption Standard, http://csrc.nist.gov/publications/PubsFIPS.html

[6] FIPS 180-4 Secure Hash Standard, http://csrc.nist.gov/publications/PubsFIPS.html

[7] FIPS 198-1 The Keyed-Hash Message Authentication Code (HMAC), http://csrc.nist.gov/publications/PubsFIPS.html

[8] FIPS 186-4 Digital Signature Standard (DSS), http://csrc.nist.gov/publications/PubsFIPS.html

[9] ANSI X9.52:1998 Triple Data Encryption Algorithm Modes of Operation, http://webstore.ansi.org/FindStandards.aspx?Action=displaydept&DeptID=80&Acro=X9&DpName=X9,%20Inc.

[10] NIST SP 800-67 Revision 1, Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher, http://csrc.nist.gov/publications/PubsFIPS.html

[9] NIST SP 800-38B, Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication, http://csrc.nist.gov/publications/PubsFIPS.html

[10] NIST SP 800-38C, Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality, http://csrc.nist.gov/publications/PubsFIPS.html

[11] NIST SP 800-38D, Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, http://csrc.nist.gov/publications/PubsFIPS.html

[12] NIST SP 800-38E, Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices, http://csrc.nist.gov/publications/PubsFIPS.html

[13] NIST SP 800-56A, Recommendation for Pair-Wise Key Establishment Schemes using Discrete Logarithm Cryptography (Revised), http://csrc.nist.gov/publications/PubsFIPS.html

[14] NIST SP 800-90A, Recommendation for Random Number Generation Using Deterministic Random Bit Generators, http://csrc.nist.gov/publications/PubsFIPS.html

[15] RSA Laboratories, “PKCS #11 v2.20: Cryptographic Token Interface Standard”, 2004. (http://www.rsasecurity.com/rsalabs/node.asp?id=2133)

[16] P. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," CRYPTO '96, Lecture Notes In Computer Science, Vol. 1109, pp. 104-113, Springer-Verlag, 1996. (http://www.cryptography.com/timingattack/)

[17] D. Boneh and D. Brumley, "Remote Timing Attacks are Practical," http://crypto.stanford.edu/~dabo/abstracts/ssl-timing.html.

[18] C. Percival, "Cache Missing for Fun and Profit," http://www.daemonology.net/papers/htt.pdf.

[19] N. Ferguson and B. Schneier, Practical Cryptography, Sec. 16.1.4 "Checking RSA Signatures", p. 286, Wiley Publishing, Inc., 2003.

