F-Secure Corporation

F-Secure® Kernel Mode Cryptographic Driver™ for Linux
FIPS 140-2 Validation Security Policy

Author: Alexey Kirichenko

Module version: 2.3.9

Document version:
F-Secure,FSCLM, FSCLM_Linux_kernel_SP.rtf,00000004

Created: May 2006
Last modified: January 2011

Abstract: This document describes the F-Secure® Kernel Mode Cryptographic Driver™ Security Policy submitted for validation, in accordance with the FIPS publication 140-2, level 1.

Introduction

The F-Secure® Kernel Mode Cryptographic Driver™ for Linux kernel (the Module) is a software module implemented as a shared library (FSCLM.KO). When loaded into computing system memory, it resides at the kernel mode level of the Linux Operating System and provides an assortment of cryptographic services that are accessible by other kernel mode drivers through an Application Programming Interface (API).

In certain cases, it is very important to have access to cryptographic services in the kernel mode. For instance, file and disk encryption products and implementations of Virtual Private Network (VPN) concept usually include kernel mode components which make extensive use of cryptographic functions, such as encryption, hashing, and random bits generation. For such a component, cryptographic service providers residing in the user mode are of little help because of a significant performance penalty associated with calling user mode functions from the kernel mode. This penalty is hardly acceptable in products operating in real time. It is also more error-prone and difficult to use user mode services from the kernel mode in a secure and reliable way. Therefore, the F-Secure Kernel Mode Cryptographic Driver, whose high performance API functions can be directly called from other kernel mode drivers, may bring considerable value to software vendors developing real-time data security products for Linux Operating System.

The Module was tested for FIPS 140-2 Level 1 requirements on Red Hat Enterprise Linux (RHEL) 5 operating system.

Overall Design and Functionality

The Module is designed and implemented to meet the Level 1 requirements of FIPS publication 140-2 when running on a GPC under Linux Red Hat Enterprise Linux v5 operating system.

The Module is written in the “C” programming language. At the source code level, we use nearly an identical set of source files to build cryptographic libraries for a number of platforms, operating systems and linkage options. The version being validated, a Linux kernel version, is a kernel object for Linux Red Hat Enterprise Linux v5 operating system.

The Module supports the FIPS approved AES (Cert. #1556), Triple DES (Cert. #1020)), SHA-1 (Cert. #1380), HMAC-SHA-1 (Cert. #908), SHA-256 (Cert. #1380), and HMAC-SHA-256 (Cert. #908) algorithms. It also provides non-FIPS approved DES, Blowfish, RC2, MD5, HMAC-MD5, RIPEMD-160, HMAC-RIPEMD-160, and passphrase-based key derivation (PBKDF2 as specified in PKCS#5) algorithms. The Module implements a high-quality cryptographically strong random Number Generator (RNG), which is compliant with the algorithm specified in Appendix 3.1 of the **FIPS PUB 186-2** document (Cert. #837).

Since the cryptographic driver is a software module that runs on a general-purpose computing systems and does not support asymmetric cryptographic methods, no special effort was taken to mitigate side-channel attacks, in particular those based on timing and power analysis and fault induction.

All the cryptographic services implemented within the Module are available only to kernel mode system drivers, which are a part of the operating system Trusted Computer Base (TCB). It is impossible to access any of the Module services directly from user mode programs. This approach is chosen, in particular, to reduce the risk of a targeted attack on the Module by malicious code.

Use of an appropriate synchronization technique in the Module helps ensure that it functions correctly when simultaneously accessed by multiple threads.

The Cryptographic Module and Cryptographic Boundary

In FIPS140-2 terms, the Module is a “multi-chip standalone module.” The F-Secure Kernel Mode Cryptographic Driver for Linux runs as a Kernel Object in any commercially available computing system under the Linux OS. A “cryptographic boundary” for the Module is defined as those applicable software and hardware components internal to a host computing system that is running the operating system.

The OS and the underlying central processing unit (CPU) hardware control access to the non-paged memory space in such a way that it is accessible only in the kernel mode. Being a kernel mode driver, the Module resides in the non-paged space. As cryptographic services provided by the Module are available only to other kernel mode drivers, we immediately see that any data passed between the Module and its clients can be accessed only in the privileged mode of the OS and never leave the cryptographic boundary.

The module provides no physical security beyond that of the physical enclosure of a “hosting” computer system.

The assumption, which we make about the operating environment of the Module, is that it is installed, initialized and used by following the rules described below in section “Roles and Services.”

The Module was internally tested by the vendor (F-Secure Corporation) on the following computing platform:

Hardware: Fujitsu Siemens Lifebook E8020
Processor: Intel Pentium M 2 GHz, 1 core
Operating System: RHEL 5, Kernel 2.6.18-164.6.1

and

Hardware: Fujitsu Siemens Scenic P300
Processor: Intel Celeron 2.66 GHz
Operating System: CentOS 5.3, Kernel 2.6.18-164.6.1

Additionally, the Module was tested by a CMVP laboratory on the following computing platforms:

Hardware: Dell SC420
Processor: Intel Celeron CPU 2.53 Ghz
Operating System: Red Hat Enterprise Linux v5

Roles and Services

The Module implements the following two roles: Crypto Officer role and User role. Since the Module is validated at security level 1, it does not provide an authentication mechanism. Hence the roles are assumed implicitly based on the services that are performed.

The two roles are defined per the FIPS140-2 standard as follows:
A **User** is any entity that can access services implemented in the Module.
A **Crypto Officer** is any entity that can access services implemented in the Module, install the Module in a device, and configure the device to ensure proper operating of the Module in the FIPS 140-2 mode of operation.
There is no **Maintenance** role.

An operator performing a service within any role can read and write security-relevant data only through the invocation of a service by means of the Module API.

The following operational rules must be followed by any user of the Module:

1. Virtual memory of the computing system must be configured to reside on a local, not a network, drive.
2. A special operating system device providing high quality randomness must be present on the computer. The Module attempts to read data from both the blocking /dev/random device, and the non-blocking /dev/urandom device to seed its RNG.

It is a responsibility of the Crypto-Officer to configure the operating system to operate securely and, whenever it is necessary, to prevent remote login. Note that the Crypto Officer must have administrative privileges in the computer system being configured.

The services provided by the Module to the User are effectively delivered through the use of appropriate API calls. In this respect, the same set of services is available to both the User and the Crypto Officer.

When the OS loader attempts to load the Module into memory, the Module runs an integrity test and a number of cryptographic functionality self-tests. If all the tests pass successfully, the Module makes a transition to “User Service” state, where the API calls can be used by other kernel mode drivers to carry out desired cryptographic services. Otherwise, the Module returns to “Uninitialized” state and the OS reports failure of the attempt to load it into memory.

The Module provides the following FIPS-approved services:
1. Cryptographic data hashing using FIPS PUB 180-2 SHA-1 and SHA-256.
2. Symmetric data encryption and decryption using FIPS PUB 197 AES and FIPS PUB 46-2 Triple-DES.
3. Random number generation using a software-based algorithm as specified in FIPS 186-2, *Digital Signature Standard (DSS*), Appendix 3.1.
4. MAC computation and verification using FIPS PUB 198 HMAC-SHA-1 and HMAC-SHA-256 algorithms (when key size is at least half of the algorithm output size).

Other non-approved services provided by the Module include:
5. Cryptographic data hashing using MD5 and RIPEMD-160 algorithms.
6. MAC computation and verification using HMAC-MD5 and HMAC- RIPEMD-160 algorithms.
7. Symmetric data encryption and decryption using Blowfish and RC2 block ciphers.
8. Passphrase-based key derivation (PBKDF2 as specified in PKCS#5) algorithm.
9. Symmetric data encryption and decryption using DES.

Non-FIPS-approved services cannot be selected if the Module is operating in accordance with FIPS 140-2, that is, in the FIPS mode of operation. The exception to this is the Passphrase-based key derivation service based on the FIPS-approved SHA-1 hash function and HMAC-SHA-1 algorithm. This service provides functionality that is not properly covered by any of the FIPS-approved algorithms at present time.

We note that the client must ensure that keys derived with PBKDF2 are only used for authentication purposes while in the FIPS mode. Such keys cannot be used for symmetric encryption/decryption when the Module is in the FIPS mode of operation.

Key Management

The Module implements a number of functions that are either used internally or exposed in the API to meet the FIPS140-2 Level 1 requirements for Key Management.

Key Generation

Keys for symmetric ciphers and HMAC algorithms can be generated by simply requesting the RNG implemented in the Module to produce a desired number of bytes. The RNG employs a FIPS-approved algorithm as specified in FIPS 186-2, *Digital Signature Standard (DSS*), Appendix 3.1. No other RNGs are used by the Module.

Intermediate key generation values are never output from the Module.

Key Distribution and Storage

All keys are processed, stored, and used in the Module only on behalf of and for immediate use by its clients, which all belong to TCB and run in the system process.
Since the current version of the Module does not support any public key methods, there is no easy way to use it for electronic key distribution in the frames of a NIST-approved key distribution protocol or for implementing standard key exchange protocols.
If, nevertheless, someone wants to use the Module API for implementing a key distribution/exchange algorithm, it is their responsibility to ensure FIPS 140-2 compliance of protocols and algorithms they implement.

The Module does not provide long-term cryptographic key storage.

Zeroization of Keys

Keys and critical security parameters in the Module can be divided into two groups: those used by the Module internally and the ones that actually belong to its clients.

The Module takes care of zeroizing all its internal keys and critical security parameters (such as the RNG internal state or various pre-computed values): (1) when those are not needed any more, (2) when the OS loader calls the Module's "unload" function, and (3) when the Module enters the error state. Also, as a precaution, the RNG internal state gets overwritten when the Module processes an unregistration request of its last client.

For the other group, when a client requests the Module to destroy a data object containing keys or critical security parameters, the Module always zeroizes all such data objects prior to freeing their memory. Also, when a client calls the "client unregistration" function, provided by the Module API, the Module zeroizes and frees memory of all data objects which are allocated and left unfreed by the client. Finally, the Module performs so-called "objects clean-up at exit." If the OS loader calls the Module's "unload" function, we check if there are any objects (like cipher or HMAC contexts) allocated and not freed by any of the clients, and we zeroize and free all such objects. This is especially

important if a fatal error occurs in the Module, or some of the clients do not have a chance to take proper care of cleaning up objects possibly containing secret information.

Protection of Keys

We rely on the OS memory management mechanism to ensure that process space of the system process, including its memory, cannot be accessed by any other process. Keys created within or passed into the Module for one user are not accessible to any other user via the Module. It is a responsibility of its clients to protect keys exported from the Module and validate keys passed into the Module.

The Module takes care of never exposing its own internal keys and critical security parameters outside, and of zeroizing those prior to exiting or freeing corresponding portions of memory. In particular, we mention the RNG state and intermediate generation values, whose disclosure or modification may compromise the security of the Module.

All dynamic memory allocations in the Module are made from the non-paged pool to ensure that blocks containing confidential data never get paged by the OS.

List of Keys stored in the module

Following keys are stored in the Module:

1. Keys for symmetric encryption/decryption algorithms:
    a. DES key
    b. Triple DES key
    c. AES key
    d. Blowfish key
    e. RC2 key

2. Keys for HMAC methods:
    a. HMAC-SHA-1 key
    b. HMAC-SHA-256 key
    c. HMAC-MD5 key
    d. HMAC-RIPEMD-160 key

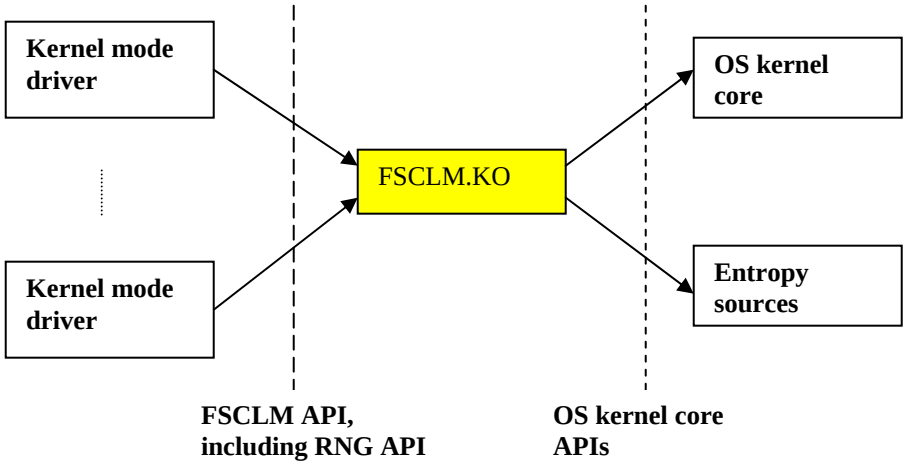
3. Key for self-integrity test:
    a. HMAC-SHA-1 key

Out of the above keys, only the HMAC-SHA-1 key used for the self-integrity test is stored across power cycles. The rest of the keys are ephemeral keys, which are zeroized before the Module exits.

Module Interfaces

Being a software module, the F-Secure Kernel Mode Cryptographic Driver defines its interfaces in terms of the API that it provides. We define Data Input Interface as all those API calls that accept, as their arguments, data to be used or processed by the Module. The API calls that return, by means of return value or arguments of appropriate types, data generated or otherwise processed by the Module to the caller constitute Data Output Interface. Control Input Interface is comprised of the call used to initiate the Module and the API calls used to control the operation of the Module. Finally, Status Output Interface is defined as the API calls, which provide information about the status of the Module.

Here is a logical block diagram of the Module's interaction with other system components (the cryptographic boundary is shown in yellow). Note that other kernel mode drivers access the RNG functionality provided by the Module through the RNG API, which is a part of the FSCLM API and described in the "RNG Functions" subsection of the "List of the API Functions, Operating Modes, Important Technical Considerations" section (p. 20). The Module accesses various entropy sources for seeding and reseeding its RNG, including /dev/random device and /dev/urandom device, through appropriate kernel core APIs.



Self-Testing

The F-Secure Kernel Mode Cryptographic Driver implements a number of self-tests to check proper functioning of the Module. This includes power-up self-tests (which are also callable on-demand) and conditional self-tests.

Power-up Self-Testing

When the Module starts loading into memory, power-up self-testing is initiated automatically. It is comprised of the software integrity test and known answer tests of cryptographic algorithms. If any of the tests fail, the Module returns to “Uninitialized” state and the OS reports failure of the attempt to load it into memory.

The following known answer tests are implemented in the Module:
· AES KAT
· Triple-DES KAT
· SHA-1 KAT
· SHA-256 KAT
· HMAC-SHA-1 KAT
· RNG KAT

The software integrity test computes DAC value by applying the HMAC-SHA-1 method, FIPS 198, to data of all the relevant sections of disk image of the Module. The test fails if the DAC value computed on the disk image of the Module does not match the original value computed on the Module by a special utility at the vendor’s site (F-Secure Corporation) and stored in a special place inside the Module.

On-Demand Self-Testing

The Module exports an API routine, “fsclm_Selftest”, which can be called to initiate the power-up self-tests. If any of the tests fail, the Module enters the error state. This error state is unrecoverable; upon entering it, the Module stops providing cryptographic services to the client.

Conditional Self-Testing

This includes continuous RNG testing. The very first output block generated by the RNG is never used for any purpose other than initiating the continuous RNG test, which compares every newly generated block with the previously generated block. The test fails if the newly generated RNG output block matches the previously generated block. In such a case, the Module enters the unrecoverable error state.

List of the API Functions, Operating Modes, Important Technical Considerations

In this section, we briefly describe the services that the Module provides and related security and usage considerations. In order to guarantee secure and robust functioning of the Module, it is important that the clients follow our recommendations as fully and precisely as possible.

The following list presents the Module API functions split into a number of groups in accordance with their functionality.

Mode of operation and Information functions

fsclm_GetModuleVersion
This routine provides the callers with the Module version information.

fsclm_GetModuleMode
This routine returns the current mode of operation of the Module.
The F-Secure Kernel Mode Cryptographic Driver supports two modes of operation: FIPS 140 mode and non-FIPS mode. Only FIPS-approved algorithms are available to the caller in FIPS 140 mode. Any attempt to use non-FIPS-approved algorithms in FIPS 140 mode results in an appropriate error code returned by the Module. It is a responsibility of client application developers to design their products in such a way that they function properly in the both modes of operation. We recommend avoiding schemes and protocols, which are based on non-selectable non-FIPS-approved algorithms in any part.

fsclm_SetModuleMode
This routine sets the mode of operation of the Module. The two options are:
FSCLM_MODE_NONFIPS - all methods included in the Module are available to the caller;
FSCLM_MODE_FIPS140 - only FIPS-approved methods are available to the caller.
Use of "fsclm_SetModuleMode" makes it easy to ensure that non-FIPS-approved algorithms are unavailable, no matter what cryptographic services the client application requests from the Module.

fsclm_GetModuleStatus
This routine returns the current status of the Module. There are five states defined in the Module Finite State Machine (FSM):
FSCLM_STATUS_UNINITIALIZED
FSCLM_STATUS_SELF_TESTING
FSCLM_STATUS_USER_SERVICE
FSCLM_STATUS_UNLOADING
FSCLM_STATUS_ERROR

fsclm_GetErrorCode
This function returns "fatal" error code if the Module is in the error state, or
FSCLM_ERROR_FATAL_NONE otherwise.

Symmetric encryption functions

The Module implements a number of symmetric ciphers, including FIPS-approved AES and Triple-DES modes. In the code, we use a layered approach based on the internal “cipher API”, which makes it very easy to exclude existing or add new ciphers if desired. The cipher modes of operation are implemented as a generic layer, so any newly included cipher can immediately be used in any of the supported modes. The Module supports the standard ECB, CBC, CFB, OFB, and Counter modes and the IWEC mode. The latter combines the ECB and Counter modes and is a good choice in certain applications.

All the encryption and decryption functions support “in-place” operations, which means that the same buffer may be used as both source and destination parameters. Note, however, that the source and destination buffers must not partially overlap. Also, in the functions accepting IV/Counter value as one of the parameters, the buffer containing that value must not overlap with the source and destination buffers.

fsclm_CipherInfo
Provides information about the specified cipher. This makes it possible to learn if the cipher is supported by the Module, if it is FIPS-approved, and what key and block sizes are supported for it.

fsclm_CipherAlloc
Allocates and initializes the cipher context object for the specified cipher in the specified mode of operation and with the specified key. Any allocated cipher object must eventually be freed by calling "fsclm_CipherFree". The Module takes care of never exposing contents of cipher objects outside and of proper zeroizing their memory when appropriate.

fsclm_CipherFree
Zeroizes and frees the memory of the specified cipher object. This routine is always available to the caller, even if the Module is in the error state.

fsclm_CipherReset
This resets the given cipher object so that it would look like a newly allocated and initialized one. The "reset" operation also zeroizes all remnants of the previous processing.

fsclm_CipherEncrypt
This encrypts the given input buffer and writes the resulting ciphertext to the given output buffer. Encryption mode and other parameters are taken from the given cipher context object.

fsclm_CipherDecrypt
This decrypts the given input buffer and writes the resulting plaintext to the given output buffer. Mode of operation and other parameters are taken from the given cipher context object.

fsclm_CipherEncryptIV
This encrypts the given input buffer and writes the resulting ciphertext to the given output buffer. The only difference between this routine and "fsclm_CipherEncrypt" is that the latter takes IV/counter information from the cipher object and updates it appropriately, while the former uses "iv" value passed to it as a parameter and updates that value (leaving IV/counter information in the cipher object intact).

fsclm_CipherDecryptIV
This decrypts the given input buffer and writes the resulting plaintext to the given output buffer. The only difference between this routine and "fsclm_CipherDecrypt" is that the latter takes IV/counter information from the cipher object and updates it appropriately, while the former uses "iv" value passed to it as a parameter and updates that value (leaving IV/counter information in the cipher object intact).

fsclm_CipherSetIV
This sets encryption or decryption IV/counter value in the specified cipher object. This value will then be used for the subsequent encryption ("fsclm_CipherEncrypt") or decryption ("fsclm_CipherDecrypt") operation respectively.
Note that the same cipher object can be used for both encryption and decryption operations, thus we maintain separate encryption and decryption IV/counter information in the cipher object.

fsclm_CipherGetIV
This copies the current encryption or decryption IV/counter value in the specified cipher object to the caller-supplied buffer.

fsclm_CipherComputeIV
Certain modes of operation of block ciphers make use of counter value. In such modes, processing of a particular block of input depends on the initial value of counter and index (or offset) of the block. (Two examples supported by the Module are Counter and IWEC modes.) If you want to perform encryption or decryption operation starting with the *n*-th block, you would need to know the corresponding counter value, and this is what this routine helps you do: given the initial counter value and the block index, it computes and writes to the caller-supplied buffer the counter value for the block.
Note that counter-based modes provide you with a random read-write access to large streams of encrypted data, the property that CBC, CFB, and OFB modes do not enjoy.

fsclm_CipherEncryptBuffer
This routine performs one-pass encryption of a given buffer, which can be a useful shortcut in certain cases. It encapsulates a number of other API calls to save the application developer effort. This call is equivalent to the following sequence:
fsclm_CipherAlloc
fsclm_CipherEncryptIV
fsclm_CipherFree

fsclm_CipherDecryptBuffer
This routine performs one-pass decryption of a given buffer, which can be a useful shortcut in certain cases. It encapsulates a number of other API calls to save the application developer effort. This call is equivalent to the following sequence:
fsclm_CipherAlloc
fsclm_CipherDecryptIV
fsclm_CipherFree

Hash functions

The Module currently implements FIPS-approved SHA-1 and SHA-256, and non-FIPS-approved MD5 and RIPEMD-160 hash functions. In the code, we use a layered approach based on the internal “hash API”, which makes it very easy to exclude existing or add new hash functions if desired.

fsclm_HashInfo
Provides information about the specified hash function. This makes it possible to learn if the hash function is supported by the Module, if it is FIPS-approved, and what its output (digest) and block sizes are.

fsclm_HashAlloc
Allocates and initializes the hash context object for the specified hash function. Any allocated hash object must eventually be freed by calling "fsclm_HashFree".
Hash objects may contain confidential information. The Module takes care of never exposing contents of hash objects outside and of proper zeroizing their memory when appropriate.

fsclm_HashFree
Zeroizes and frees the memory of the specified hash object. This routine is always available to the caller, even if the Module is in the error state.

fsclm_HashReset
This resets the given hash context object so that it would look like a newly allocated and initialized one. It is useful when you want to use the same hash function for computing hash values (also called *digests*) of multiple data blocks.
The "reset" operation also zeroizes all remnants of the previous processing.

fsclm_HashUpdate
This updates the given hash context with the given input.
When you need to compute digest of a data stream which comes in a number of portions (or when you want to split a very long stream in a number of pieces), you can simply feed such portions to "fsclm_HashUpdate" one by one. The resulting digest value will be identical to what you would get if passing the entire stream as a single buffer.
Note that in order to obtain digest value of your data, any sequence of calls to "fsclm_HashUpdate" must eventually be followed by a call to "fsclm_HashFinal".

fsclm_HashFinal
This function completes computation of hash value of a data stream, which has been processed by calls to "fsclm_HashUpdate" function. The resulting digest is written to a caller-supplied buffer.
Note that after "fsclm_HashFinal" has been called for a hash object, the object should not be used for any further operations until you call "fsclm_HashReset" for it. After resetting, you may start computation of hash value for a new data stream.

fsclm_HashOfBuffer
This routine computes digest of a given buffer, which can be a useful shortcut in certain cases. It encapsulates a number of other API calls to save the application developer effort. This call is equivalent to the following sequence:

fsclm_HashAlloc
fsclm_HashUpdate
fsclm_HashFinal
fsclm_HashFree

HMAC functions

The Module clients can use HMAC methods based on any hash function that is implemented in the Module. By specifying the ID of a hash function of your choice, you fully specify the HMAC algorithm that you want to use. To obtain information about parameters of a particular HMAC algorithm, simply call "fsclm_HashInfo" for the corresponding hash function.

fsclm_HMACAlloc
Allocates and initializes the context object for the HMAC algorithm based on the specified hash function, and with the specified key. Any allocated HMAC object must eventually be freed by calling "fsclm_HMACFree".
HMAC objects contain keying information derived from keys passed to the HMAC allocation function and may contain other confidential information. The Module takes care of never exposing contents of HMAC objects outside and of proper zeroizing their memory when appropriate.

fsclm_HMACFree
Zeroizes and frees the memory of the specified HMAC object. This routine is always available to the caller, even if the Module is in the error state.

fsclm_HMACReset
This resets the given HMAC context object so that it would look like a newly allocated and initialized one. It is useful when you want to use the same HMAC function, possibly with a different key, for computing message authentication code (MAC) values of multiple data blocks.
The "reset" operation also zeroizes all remnants of the previous processing.

fsclm_HMACUpdate
This updates the given HMAC context with the given input.
When you need to compute MAC of a data stream which comes in a number of portions (or when you want to split a very long stream in a number of pieces), you can simply feed such portions to "fsclm_HMACUpdate" one by one. The resulting MAC value will be identical to what you would get if passing the entire stream as a single buffer.
Note that in order to obtain MAC value of your data, any sequence of calls to "fsclm_HMACUpdate" must eventually be followed by a call to "fsclm_HMACFinal".

fsclm_HMACFinal
This function completes computation of MAC value of a data stream, which has been processed by calls to "fsclm_HMACUpdate" function. The resulting MAC is written to a caller-supplied buffer.
Note that after "fsclm_HMACFinal" has been called for an HMAC object, the object should not be used for any further operations until you call "fsclm_HMACReset" for it. After resetting, you may start computation of MAC value for a new data stream (possibly using a different key).

fsclm_HMACOfBuffer
This routine computes MAC value of a given buffer, which can be a useful shortcut in certain cases. It encapsulates a number of other API calls to save the application developer effort. This call is equivalent to the following sequence:
fsclm_HMACAlloc
fsclm_HMACUpdate

fsclm_HMACFinal
fsclm_HMACFree

RNG functions

The RNG implemented in the Module is based on hybrid architecture. It uses a one-way output function on top of the well-known “entropy pool” scheme. The design is FIPS-compliant as the output algorithm is the one specified in Section 3.1, Appendix 3 of **FIPS PUB 186-2** document, with the function G constructed from the SHA-1 as specified in Section 3.3, Appendix 3 of the same document.

The RNG is initialized when the Module gets loaded into memory. During the initialization phase, various system and hardware parameters and statistics are collected and mixed in the RNG pool with the SHA-1 transform function to achieve a good diffusion of “entropy” bits. Seeding/reseeding code for each supported platform resides in the respective platform-specific source file.

fsclm_PrngDeepPoll
Invokes platform-specific “deep” polling for entropy (i.e., hard-to-predict bits) to achieve good-quality seeding of the RNG. This deep polling gets called automatically occasionally during the entire lifetime of the Module. Also, the function is called at the RNG initialization time.
The main purpose of this function is to help maintain the RNG pool in a state, which is infeasible to guess for the adversary.

fsclm_PrngAddNoise
This exclusive-ORs bytes from the given buffer with the RNG pool content and serves the purpose of adding unpredictability to the RNG state. (We leave it up to the client whether to use this function or not as the automatic RNG seeding in the Module should be good enough to prevent the adversary from guessing the RNG state or any of the output values.)
The exclusive-OR operation cannot force the RNG in a weaker state because it obviously cannot reduce the pool data entropy.

fsclm_PrngMixPool
Mixes (i.e., cryptographically processes) the RNG pool. The mixing operation is based on the SHA-1 transform function. It provides good “entropy” diffusion and is irreversible.
This function gets called automatically at the initialization time and then regularly during the entire lifetime of the Module.

fsclm_PrngGetBytes
This routine writes to the caller-supplied buffer the requested number of RNG-produced bytes. Although what the generated bytes will be used for is entirely up to the caller, we recommend calling this function if you need to generate:
- any keying material (in both symmetric and asymmetric settings)
- IV or initial counter values used in many popular methods (e.g., modes of operation of block ciphers)
- padding bytes for various cryptographic schemes
- random nonces and challenges required in many cryptographic protocols (e.g., authentication protocols)
- salts to be combined with passphrases in passphrase-based key derivation algorithms
- random values for probabilistic cryptographic algorithms (e.g., signing with DSA)

We stress that it is a responsibility of the client to protect bytes provided by the Module RNG (in particular, from being exposed to the adversary).

fsclm_PrngGetParameters
Fills in the fields of a caller-supplied structure with the current values of the RNG object parameters. The function that sets the RNG parameters, “fsclm_PrngSetParameters”, is unavailable in the API of the F-Secure Kernel Mode Cryptographic Driver for the time being. This is mostly due to the fact that in the kernel mode, a single instance of the Module may serve to multiple callers – kernel mode drivers, so the RNG object is shared between all the callers.

Client registration functions

fsclm_RegisterCaller

Prior to using any of the cryptographic services provided by the Module, the clients must register to the Module. Successful registration results in a “reference” token returned to the client. That token should then be passed as a parameter to almost all the API functions the client calls. (A small number of information functions do not have the “caller reference” argument and can be used without registering.)

fsclm_UnregisterCaller

When the client does not need the Module services any longer, it must call the unregistration function. Such a call results in freeing the memory associated with the client. All cryptographic objects allocated and not freed by the client will be zeroized and freed by the Module. This helps ensure no confidential data will be left in memory.

We strongly recommend to our clients to ensure they eventually unregister with the Module. (Note that it may be insufficient to simply put the unregistration function in the "unload" function of your driver, as the latter function does not get called by the OS loader when the system is about to be shut down. Thus, you may want to process "system shutdown" notification sent by the OS to take your chance to unregister.)

The unregistration routine is always available to the client, even if the Module is in the error state. In fact, we recommend calling it as soon as you found out that the Module had entered the error state.

Other functions

fsclm_Selftest
Calling this routine makes the Module run a number of self-tests. This on-demand self-testing includes self-integrity test, Known Answer Tests of cryptographic algorithms, and, optionally, the set of RNG statistical tests (as specified in the FIPS 140-2 document). If any of the tests fail, the Module enters the error state, which means that its cryptographic services become unavailable to the clients. To use the services again, the user will need to restart the Module.

fsclm_DeriveSymmetricKey
This routine implements the passphrase-based key derivation function specified in PKCS#5 (PBKDF2). The implementation uses HMAC-SHA1 as a PRF.
The two main goals of this key derivation algorithm are:
- preventing the adversary from compiling a universal dictionary of passphrases and precomputing the corresponding keys (achieved by using so-called “salt”, whose presence in the algorithm results in a very large number of keys that correspond to each passphrase)
- making exhaustive search attacks much more computationally expensive, which is especially important in the case of “weak” passphrases (achieved by iterating the key derivation function many times and recursively)
We stress that it is a responsibility of the client to protect keys derived by this routine (in particular, from being exposed to the adversary). This is a non-Approved service.

fsclm_OverwriteMemory
This function can be used for overwriting a given block of memory with a bit stream that enjoys good statistical properties (i.e., appears as a Binary Symmetric Source output).
We use it internally to overwrite portions of memory that may contain confidential data.
Also, this function can (and should !) be used instead of the RNG to produce random-looking bits when we do not care about “cryptographic quality”. A typical example is generating “witnesses” for probabilistic primality testing.

fsclm_GetBase64Length
Clients should call this routine prior to calling “fsclm_EncodeBase64” to determine size of the buffer that Base64 encoded data will be written to. Values of the encoding option arguments passed to “fsclm_GetBase64Length” must be identical to the ones subsequently passed to “fsclm_EncodeBase64”.

fsclm_EncodeBase64
Given an input buffer, this routine encodes the data in Base64 format. The client can specify desired line length and ending for the encoded byte stream.

fsclm_DecodeBase64
This routine transforms a given Base64 encoded byte stream to the original (raw) form.

Detailed description of the Module API can be found in the Module public header file, FSCLM.H.

We conclude this section by listing a number of recommendations aimed at helping the Module clients avoid security-related and technical problems when implementing data security products.

- Prior to freeing any memory blocks that may contain critical security parameters or other confidential data, take care of zeroizing them properly. When you free an object allocated by the Module (for example, a symmetric cipher context) by calling an appropriate FSCLM API function, the Module zeroizes the object memory. The client applications are responsible for zeroizing any other memory blocks, in particular, those intermediate variables containing keying or otherwise confidential data.

- It is a responsibility of the clients to ensure they work with cryptographic objects allocated by the Module in a multi-threading safe way. Please keep in mind that the Module provides no synchronisation for accessing such objects concurrently by multiple threads of the client applications.

- All dynamically allocated memory blocks that may contain critical security parameters or other confidential data should be allocated from the non-paged pool. We follow this rule in the Module code as this is the best way to ensure that blocks containing confidential data never get paged by the OS.