MOVEit Crypto FIPS 140-2 Security Policy

Author: Mark Riordan, Standard Networks, Inc.

Module version: 1.1.0.0

Document Revision Date: 2003-11-17

Standard Networks, Inc
344 S Yellowstone Dr
Madison, WI 53705
608.227.6100
http://www.stdnet.com

Table of Contents

Introduction

This document describes the security policy for MOVEit Crypto, to meet the FIPS 140-2 requirements. This document is non-proprietary and may be distributed freely.

MOVEit Crypto is a compact and fast encryption library that provides an API featuring the latest NIST-approved encryption, hashing, and pseudo-random number generation algorithms. The easy-to-use programming interface allows applications to be written without special code for block size and padding mode.

MOVEit Crypto is a member of the MOVEit security and file transfer product family. For more information, see http://www.stdnet.com/moveit.

Cryptographic Module and Cryptographic Boundary

MOVEit Crypto is a dynamically linked library that provides encryption services to applications. Separate versions exist for Microsoft Windows operating systems, and the Linux operating system. (The Linux version also runs on FreeBSD.) MOVEit Crypto is supported on Windows 95/98/ME/NT 4.0/2000/XP/2003, Linux, and FreeBSD, and was tested on Windows 2000 and Red Hat Linux 9.0 in single-user mode.

MOVEit Crypto is purely a software product. In FIPS 140-2 terms, MOVEit Crypto is a multi-chip standalone module, consisting of the file MOVEitCrypto.dll (on Windows) or libmoveitcrypto.so.1.1 (on Linux and FreeBSD). It is intended to meet FIPS 140-2 security level 1.

The cryptographic boundary for MOVEit Crypto is defined as the enclosure of the computer on which the cryptographic module is installed. As a software product, MOVEit Crypto provides no physical security by itself. The computer itself must be appropriately physically secured.

Single Operator Only

MOVEit Crypto runs in Single User Mode. Multiple concurrent operators are not supported.

Roles

MOVEit Crypto implements the Cryptographic Officer and User roles. There is no Maintenance role.

The module does not provide any identification or authentication of its own, and does not by itself provide a way to restrict a user to one role or the other.

User

The user is any entity that can access services provided by the module. All services are available to a user. See the Services section of this document. The User role is implicitly selected when a process calls any API function in the module.

Cryptographic Officer

The cryptographic officer is any entity that can install the module onto the computer system, configure the operating system, or access services provided by the module. The cryptographic officer may access all services, the same as a user.

The cryptographic officer has no special access to any keys or data.

The cryptographic officer role is implicitly selected when installing the module or configuring the operating system.

Installation is accomplished by running an installation program. The cryptographic officer must have permission to write the file constituting the module into the installation directory; typically, this requires administrator access within the operating system. On Windows, the installation directory is the Windows System directory and the cryptographic officer typically should be a member of the Administrators group. On Linux and FreeBSD, the installation directory is /usr/lib and the cryptographic officer should have root access.

The cryptographic officer should ensure that the operating system runs in single user mode. This is automatic for Windows 95/98/ME. For Windows NT/2000/XP/2003, this can be done by disabling the Server and RunAsService services using the Control Panel. The paging file must be configured to be on a local drive, not a network drive.

For Linux and FreeBSD, a running system can be placed into single-user mode via the command `/sbin/shutdown now`. Alternatively, Linux can be configured to always boot into single-user mode. This is done by editing the file /etc/inittab. Change the line that looks like `id:`*x*`:initdefault` (where *x* is a number, usually 3, 4, or 5) to `id:1:initdefault`. When the system is rebooted, it will run in single-user mode.

Services

MOVEit Crypto's services consist entirely of an Application Programming Interface (API), which is a collection of functions that can be called from an operating system process. All services provided by the module are described in this section, and both the User and Crypto-officer roles have access to all of these services.

Overview of Functions and Modes of Operation

FIPS-140-2 Approved Modes of Operation

MOVEit Crypto provides the following FIPS 140-2 approved algorithms:

Encryption: AES, as described in FIPS PUB 197, with 128-, 192-, and 256-bit keys

Hashing: SHA-1, as described in FIPS PUB 180-1

Keyed hashing: HMAC-SHA-1, as described in FIPS PUB 198.

Pseudo-random number generation: RNG using the FIPS-approved algorithm described in FIPS PUB 186-2, Appendix 3.1

MOVEit Crypto is operating in a FIPS-approved mode when its implementation of one of the above algorithms is being used.

FIPS-140-2 Non-Approved modes of Operation

MOVEit Crypto provides the following algorithms that are not approved by FIPS 140-2:

Hashing: RSA Data Security, Inc. MD5 Message-Digest Algorithm, as described in Internet RFC 1321

Keyed hashing: HMAC-MD5, as described in Internet RFC 2104

MOVEit Crypto is **not** operating in a FIPS-approved mode when its implementation of one of these algorithms is being used.

Thus, the indication of when the module is operating in a non-FIPS-approved mode is when the algorithm MC_ALG_MD5 is being used via MCHashInit() or MCKeyedHashInit(). The module is in FIPS-approved mode at all other times.

Function Descriptions

All functions have names starting with the string MC. The functions are implemented using a C language interface, in order to make them available to the greatest number of programming languages. Although the documentation below is provided in C language terms, the functions can be called from any language that can access a dynamically-linked library.

In the function descriptions below, a C-style function prototype is followed by a description of the parameters and return value. Input parameters are marked with [in], output parameters are marked with [out], and parameters used for both input and output are marked with [in, out].

Data Types

The following data types are used throughout the API. The definitions of these types and their values are provided in a separate source code header file, MOVEitCrypto.h.

MC_CONTEXT

MC_CONTEXT is a data type passed to nearly all functions in the API. It stores information on pending cryptographic operations. The individual fields in this data type are not exposed to the user. MC_CONTEXT stores all key information and any partial plaintext or ciphertext blocks. Only the user-provided MC_CONTEXT buffer is used to store this information; the module does not maintain its own copy. The user must pass the same MC_CONTEXT buffer for all operations on a given message.

MC_ERROR

MC_ERROR is an enumerated type used as the return type by most functions. It specifies the error returned by a function. The possible values are:

Value	Meaning
MC_ERROR_NONE	Success. This value is guaranteed to be zero for all future versions of the module, so an application can test against 0 to determine whether a function succeeded.
MC_ERROR_DISABLED	MOVEit Crypto has been disabled due to failing a self-test.
MC_ERROR_UNLICENSED	MOVEit Crypto has been disabled because a valid license key has not been supplied.
MC_ERROR_BAD_CONTEXT	The context is invalid.
MC_ERROR_BAD_ALGID	The algorithm identifier is invalid.
MC_ERROR_BAD_MODE	The cipher mode parameter is invalid.
MC_ERROR_BAD_COUNT	A count parameter is invalid; for instance, a count is negative.
MC_ERROR_BAD_DATA	The data being encrypted or decrypted is invalid. For

	example, the last block of a padded message being decrypted contains invalid padding.
MC_ERROR_MISC	A miscellaneous error has occurred.

MC_ALG

MC_ALG is an enumerated type that specifies an algorithm.

Value	Meaning
MC_ALG_AES_128	AES cipher with 128-bit keys and 16-byte blocks.
MC_ALG_AES_192	AES cipher with 192-bit keys and 16-byte blocks.
MC_ALG_AES_256	AES cipher with 256-bit keys and 16-byte blocks.
MC_ALG_SHA1	SHA-1 hash function.
MC_ALG_MD5	MD5 hash function.

MC_CIPHER_MODE

MC_CIPHER_MODE is an enumerated type that specifies a cipher mode.

Value	Meaning
MC_CIPHER_MODE_ECB	Electronic Code Book mode, in which each block is enciphered individually.
MC_CIPHER_MODE_CBC	Cipher Block Chaining mode, in which each block of plaintext is XORed with the previous ciphertext block (or with an initialization vector) prior to encryption.

MC_PADDING

MC_PADDING is an enumerated type that specifies the padding mode of a block cipher.

Value	Meaning
MC_PADDING_NONE	No padding is done to the last cipher block. The input must be a multiple of the block size.
MC_PADDING_RFC2040	The last block of a message is padded to fill a complete block. If the message is a multiple of the block size, an entire block of padding is added to the ciphertext. Each byte of padding is the number of unused bytes at the end of the block.

MC_STATE

MC_STATE is an enumerated type that describes the state of MOVEit Crypto for the current process.

Value	Meaning
MC_STATE_DISABLED	The module is disabled. The process has not attached to the DLL.
MC_STATE_SELF_TEST	The module is undergoing power-up self-test and is not available for cryptographic operations. (Note: this state is strictly internal; MOVEit Crypto will never return this state to the user because it is not possible to be attached to the module in this state.)
MC_STATE_UNLICENSED	The module has passed self-test, but no valid license key has been entered.
MC_STATE_OPERATE	The module is available for all services.
MC_STATE_ERROR	An error has occurred during operation. Cryptographic services are not available.

Initialization Functions

MCSetLicenseKey

Checks the caller-supplied license key, and enables the module if it's valid. This function must be called with a valid license key before any cryptographic services can be used.

*MC_ERROR MCSetLicenseKey(const char *szLicenseKey)*

szLicenseKey	[in] is a vendor-supplied zero-terminated ASCII license key to activate the product.
Returns	an MC_ERROR code (0 for success). If the module is in an error state, the function returns an error without checking the key. Otherwise, it enables or disables the module depending whether the key is valid.

Encryption / Decryption Functions

MCEncryptInit

Initializes a context for encryption.

*MC_ERROR MCEncryptInit(MC_CONTEXT *context, MC_ALG AlgID, MC_CIPHER_MODE cipherMode, MC_PADDING padding, void *key, void *IV)*

context	[out] points to a context to initialize.
AlgID	[in] is the encryption algorithm ID (MC_ALG_xxx value).
cipherMode	[in] specifies ECB vs. CBC (MC_CIPHER_MODE_xxx value).
padding	[in] is the padding type (MC_PADDING_xxx value).
key	[in] is the key. The size is implied by AlgID.

IV	[in] is the initialization vector. A NULL pointer implies a vector of all zeros. The size is always the blocksize of the cipher. The IV is ignored if the mode is ECB.
Returns	an MC_ERROR code (0 for success). If success, context has been initialized, else it's been zeroed.

MCEncrypt

Encrypts a buffer of bytes. This function can be called repeatedly with buffers of any size.

*MC_ERROR MCEncrypt(MC_CONTEXT *context, void *inbuf, int nBytesIn, void *outbuf, int *pnBytesOut)*

context	[in, out] points to a context that's been initialized.
inbuf	[in] is a buffer of plaintext.
nBytesIn	[in] is the number of plaintext bytes. This can be any number >= 0. If necessary, MOVEit Crypto will store a partial block in the context.
outbuf	[out] may contain some output bytes.
pnBytesOut	[out] points to the number of output bytes. This may be more or less than nBytesIn, due to internal buffering of blocks. It will not be more than 16 greater than nBytesIn.
Returns	an MC_ERROR code (0 for success). If failure, context has been zeroed.

MCEncryptFinal

Finishes the encryption of a message.

*MC_ERROR MCEncryptFinal(MC_CONTEXT *context, void *outbuf, int *pnBytesOut)*

context	[in, out] points to a context that's been initialized.
outbuf	[out] may contain some output bytes, depending on the padding mode and whether MOVEitCrypto has had to buffer any bytes internally during previous MCEncrypt calls.
pnBytesOut	[out] points to the number of output bytes. It will not be more than 16, and may be as small as 0 if no internal buffering was done by previous calls.
Returns	an MC_ERROR code (0 for success). If failure, context has been zeroed.

MCDecryptInit

Initializes a context for decryption.

*MC_ERROR MCDecryptInit(MC_CONTEXT *context, MC_ALG AlgID, MC_CIPHER_MODE cipherMode, MC_PADDING padding, void *key, void *IV)*

context	[out] points to a context to initialize.
AlgID	[in] is the encryption algorithm ID (MC_ALG_xxx value).
cipherMode	[in] specifies ECB vs. CBC (MC_CIPHER_MODE_xxx value).
padding	[in] is the padding type (MC_PADDING_xxx value).
key	[in] is the key. The size is implied by AlgID.

IV	[in] is the initialization vector. A NULL pointer implies a vector of all zeros. The size is always the blocksize of the cipher. This is ignored if the mode is ECB.
Returns	an MC_ERROR code (0 for success). If success, context has been initialized, else it's been zeroed.

MCDecrypt

Decrypts a buffer of bytes. This function can be called repeatedly with any size input. If padding is in effect, the function does not decrypt the last block's worth of bytes, because it might be the last block, which would contain padding that needs to be handled differently.

*MC_ERROR MCDecrypt(MC_CONTEXT *context, void *inbuf, int nBytesIn, void *outbuf, int *pnBytesOut)*

context	[in, out] points to a context that has been initialized for decryption.
inbuf	[in] is a buffer of encrypted bytes.
nBytesIn	[in] is the number of input bytes. It can be 0, and does not need to be a multiple of the blocksize.
outbuf	[out] may contain some decrypted plaintext.
pnBytesOut	[out] points to the number of decrypted bytes that have been placed in outbuf. One block's worth of plaintext may be buffered in the context, so you may get up to 15 more or fewer output bytes than input bytes.
Returns	an MC_ERROR code (0 for success). If failure, context has been zeroed.

MCDecryptFinal

Finishes the decryption process by decrypting the last buffered block of bytes and zeroing the context.

*MC_ERROR MCDecryptFinal(MC_CONTEXT *context, void *outbuf, int *pnBytesOut)*

context	[in, out] points to a context that has been initialized for decryption.
outbuf	[out] may contain decrypted bytes.
pnBytesOut	[out] points to the number of decrypted bytes, which will be from 0 to 16.
Returns	an MC_ERROR code (0 for success). context has been zeroed.

Hashing Functions

MCHashInit

Initializes a context for hashing.

*MC_ERROR MCHashInit(MC_CONTEXT *context, MC_ALG AlgID)*

context	[out] points to the context to initialize.
AlgID	[in] is the hash algorithm to use (MC_ALG_xxx value)
Returns	an MC_ERROR code (0 for success). If success, context has been initialized, else the hash portions have been zeroed.

MCHashUpdate

Hashes some data into an initialized context.

*MC_ERROR MCHashUpdate(MC_CONTEXT *context, void *inbuf, int nBytesIn)*

context	[in, out] points to the context.
inbuf	[in] points to the bytes to hash.
nBytesIn	[in] is the number of bytes to hash.
Returns	an MC_ERROR code (0 for success). If success, context has been updated with the bytes, else the hash portions of the context have been zeroed.

MCHashFinal

Completes the hash process, producing the final hash.

*MC_ERROR MCHashFinal(MC_CONTEXT *context, void *outbuf, int *pnBytesOut)*

context	[in, out] points to the context into which data has been hashed.
outbuf	[out] contains the hash. On input, this buffer must be big enough to hold the hash. Currently, the largest hash size implemented is 20 bytes.
pnBytesOut	[out] points to an integer which has been set to the hash size in bytes. This is provided so you don't have to know the exact hash size of the algorithm you chose.
Returns	an MC_ERROR code (0 for success). In all cases, the hash portion of context has been zeroed.

MCKeyedHashInit

Initializes a context for a keyed hash.

*MC_ERROR MCKeyedHashInit(MC_CONTEXT *context, MC_ALG AlgID, void *key, int nKeyBytes)*

context	[in, out] points to a context to initialize.
AlgID	[in] is the hash algorithm on which to base the keyed hash.
key	[in] points to the key.
nKeyBytes	[in] is the number of bytes in the key.
Returns	an MC_ERROR code (0 for success). If success, context has been initialized, else it has been zeroed.

MCKeyedHashUpdate

Hashes data into a context. The algorithm is the one described in Internet RFC 2202.

*MC_ERROR MCKeyedHashUpdate(MC_CONTEXT *context, void *inbuf, int nBytesIn)*

context	[in, out] is a context that has been initialized by MCKeyedHashInit.
inbuf	[in] points to bytes which will be hashed.
nBytesIn	[in] is the number of bytes to hash.

Returns	an MC_ERROR code (0 for success). If success, context reflects the input bytes, else it has been zeroed.

MCKeyedHashFinal

Completes the keyed hashing process, producing the final hash.

*MC_ERROR MCKeyedHashFinal(MC_CONTEXT *context, void *outbuf, int *pnBytesOut)*

context	[in, out] is the context into which bytes have been hashed by MCKeyedHashUpdate.
outbuf	[out] contains the hash. On input, the buffer must be big enough to hold the hash.
pnBytesOut	[out] points to an integer which has been set to the hash size in bytes. This is provided so you don't have to know the exact hash size of the algorithm you chose.
Returns	an MC_ERROR code (0 for success). context has been zeroed.

Pseudo-Random Number Generation Functions

MCRNGInit

Initializes a context for random number generation.

*MC_ERROR MCRNGInit(MC_CONTEXT *context)*

context	[in, out] points to a context to initialize.
Returns	an MC_ERROR code (0 for success). If failure, the context has been zeroed.

MCRNGUpdate

Seeds the random number generator with external values. This routine can be called repeatedly, at any time after initialization, to seed the generator with additional input. If MCRNGUpdate is not called prior to MCRNGGenerate, the numbers generated by MCRNGGenerate will be a specific, predictable sequence.

*MC_ERROR MCRNGUpdate(MC_CONTEXT *context, void *seedBytes, int nBytesIn)*

context	[in, out] points to a context that has been initialized.
seedBytes	[in] is a buffer with seed information. For instance, you might use the time-of-day, the thread id, the amount of free disk space, etc.
nBytesIn	[in] is the number of bytes in seedBytes.
Returns	an MC_ERROR code (0 for success). If failure, the context has been zeroed. Otherwise, the seed information has been merged into the appropriate portion of the context.

MCRNGGenerate

Generates a user-specified number of pseudo-random bytes.

*MC_ERROR MCRNGGenerate(MC_CONTEXT *context, int nBytesDesired, void *bufout)*

context	[in, out] points to an context that has been initialized and, preferably, also seeded via MCRNGUpdate.
nBytesDesired	[in] is the number of random bytes desired.
bufout	[out] holds the generated bytes.
Returns	an MC_ERROR code (0 for success). If failure, the context has been zeroed.

MCRNGFinal

Finalizes the random number generation process. This simply zeros the context.

*MC_ERROR MCRNGFinal(MC_CONTEXT *context)*

context	[out] is a context to zeroize.
Returns	MC_ERROR_NONE.

Miscellaneous Functions

MCGetState

Returns the current state. This is the "Show Status Service."

MC_STATE MCGetState()

Returns	the current state, as a MC_STATE_xxx value. Note that not all states can be returned, because in some states, the module cannot be attached and therefore the function cannot be called. This function works even if the module is in an error or unlicensed state.

MCZeroizeContext

Zeroizes the context.

*MC_ERROR MCZeroizeContext(MC_CONTEXT *context)*

context	[out] has been zeroed.
Returns	an MC_ERROR code (0 for success).

MCGetVersion

Returns the version number of the module.

*char * MCGetVersion()*

Returns	a pointer to a zero-terminated ASCII string in the form a.b.c.d, where a, b, c, and d are integers from 0 to 65535. This function returns the version even if the module is unlicensed or in an error state, in order to assist the user in problem resolution.

MCSelfTest

Performs a self-test of the module--the same self-test as performed at power-up. This is the "Self-Test Service."

MC_ERROR MCSelfTest()

Returns	an MC_ERROR code (0 for success). If the test fails, the module has entered the error state.

C++ Wrapper Class

For the convenience of C++ developers, the authors of MOVEit Crypto have provided a simple C++ class that acts as a very thin wrapper to the C-based API. This class, CMOVEitCrypto, contains methods with the same names and parameters as the MCxxx functions described above, except that there is no **context** parameter, and the letters "MC" are stripped from the names of the functions. The context is maintained in a member variable, which is automatically zeroed in the destructor.

CMOVEitCrypto is defined entirely within MOVEitCrypto.h, and is therefore **not part of the cryptographic module**. Thus, the CMOVEitCrypto class is not validatable.

The methods of this wrapper class are:

```
CMOVEitCrypto();   // Constructor
~CMOVEitCrypto();  // Destructor

MC_ERROR EncryptInit(MC_ALG EncAlgID,
  MC_CIPHER_MODE cipherMode, MC_PADDING padding, void *key, void *IV);
MC_ERROR Encrypt(void *inbuf, int nBytesIn, void *outbuf, int *pnBytesOut);
MC_ERROR EncryptFinal(void *outbuf, int *pnBytesOut);
MC_ERROR DecryptInit(MC_ALG EncAlgID,
  MC_CIPHER_MODE cipherMode, MC_PADDING padding, void *key, void *IV);
MC_ERROR Decrypt(void *inbuf, int nBytesIn, void *outbuf, int *pnBytesOut);
MC_ERROR DecryptFinal(void *outbuf, int *pnBytesOut);

MC_ERROR HashInit(MC_ALG HashAlgID);
MC_ERROR HashUpdate(const void *inbuf, int nBytesIn);
MC_ERROR HashFinal(void *outbuf, int *pnBytesOut);

MC_ERROR KeyedHashInit(MC_ALG HashAlgID, const void *key, int nKeyBytes);
MC_ERROR KeyedHashUpdate(const void *inbuf, int nBytesIn);
MC_ERROR KeyedHashFinal(void *outbuf, int *pnBytesOut);

MC_ERROR RNGInit();
MC_ERROR RNGUpdate(void *seedBytes, int nBytesIn);
MC_ERROR RNGGenerate(int nBytesDesired, void *bufout);
MC_ERROR RNGFinal();

MC_ERROR ZeroizeContext();
MC_STATE GetState();
char *GetVersion();
MC_ERROR SetLicenseKey(const char *szLicenseKey);
MC_ERROR SelfTest();
```

Key Management

MOVEit Crypto performs limited key management, as described below.

Because the module is a dynamically-linked library, each process requesting access is provided its own instance of the module. Each process has full access to all the keys and data within the module.

The module contains only keys or data placed into the module via the services described in this document. No keys or data are automatically maintained by the module, or maintained after a process detaches from the module.

Key Generation

The module does not provide key generation. All keys must be entered by the user.

Key Material and Key Storage

The module does not provide any persistent storage of key material. Keys are entered by the user only via API calls. Key material is stored in the context, which is maintained in a user-supplied data structure passed in each API call. No key material is maintained inside the module between API calls. All key material is passed into and out of the module in plaintext form.

The only key material used by the module outside of the user-supplied context is that stored temporarily in local variables on the stack. Any local variables containing sensitive information are zeroed by the module before a function call returns to the user's code.

The module relies upon operating system memory protection to prevent processes from accessing each other's key material. To ensure that other processes cannot access keys and data, the caller must not use shared memory. Also, the operating system page file must not be configured to reside on a network drive.

Key Zeroization

Each MOVEit Crypto API call zeros the context before returning to the user, if the context is no longer needed (as in the Final calls), or if an error occurs. Additionally, the user may call MCZeroizeContext at any time to zero a context. Ordinarily, this is not necessary, since the user will normally call one of the Final functions at the end of processing.

Self-Tests

As required by FIPS 140-2, MOVEit Crypto automatically performs both power-up self-tests, and for certain algorithms, continuous self-tests during operation.

Power-up Self-Tests

For this module, "power-up" is when a process attempts to attach to the module. At this time, MOVEit Crypto performs these types of tests:

Software integrity test. MOVEit Crypto computes a keyed hash of the module and compares it to an embedded keyed hash that was placed into the module when it was produced. If the computed keyed hash differs from the keyed hash hard-coded into the module, the test fails. The HMAC-SHA-1 algorithm described in FIPS PUB 198 is used.

Known Answer Test. MOVEit Crypto performs known answer tests for AES, SHA-1, HMAC-SHA-1, FIPS PUB 186-2 Appendix 3.1 random number generation, and the unapproved algorithms MD5 and HMAC-MD5. If the computed result differs from the expected result hard-coded into the module, the test fails. The random number generation test is performed by entering a fixed seed and comparing the resulting pseudo-random numbers to a known result. MOVEit Crypto also performs an encrypt/decrypt test in which a buffer is encrypted and decrypted in chunks. The decrypted result is compared to the original plaintext. If the decrypted result is different than the original plaintext, the test fails.

If any of these tests fails, MOVEit Crypto returns to the disabled state, and in the case of the Windows version, refuses to allow the process to attach to the module. Thus, the cryptographic services are not available. To correct the problem, the cryptographic officer should reinstall the module.

Conditional Self-Tests

During operation, MOVEit Crypto performs one type of ongoing test. Whenever a block of pseudo-random bytes is generated as a result of a call to MCRNGGenerate, the block is compared to the previous block. If the two match, the module enters the error state, and will no longer perform any cryptographic services. (Certain non-cryptographic utility functions remain available, namely MCGetState and MCGetVersion.) Once the module is in the error state, it remains in the error state until the process detaches from the module. To correct the problem, a process should detach from and reattach to the module.

User Guidance

Choice of Algorithms and Key Lengths

The recommended values for AES encryption parameters are:

Key size	256 bits. There is only a small performance penalty associated with longer AES key sizes. Encryption with 256-bit keys is about 25% slower than with 128-bit keys.
Cipher mode	CBC. CBC is much more secure than ECB, as it hides plaintext patterns from attackers. The extra performance overhead of CBC is negligible.
Padding	Application dependent; RFC 2040 padding is more convenient for most applications, because it does not force you to restrict your messages to multiples of the block size. The extra performance overhead of RFC 2040 padding is minor.

For example:

```
MC_ERROR mcerror = MCEncryptInit(&context, MC_ALG_AES_256, MC_CIPHER_MODE_CBC,
   MC_PADDING_RFC2040, key, IV);
```

Pseudo-Random Number Generator Seeds

A calling application can request the module's pseudo-random number generator (RNG) whenever it requires random data. If the RNG is used for key generation or other cryptographic operations, it is important that the generator be seeded with unpredictable values. There are many sources of pseudo-random data on a computer, including the time-of-day, the number of milliseconds since boot, the amount of time it takes to create a thread, the contents and attributes of files, the values of performance counters, and even the hardware RNG on some Intel Celeron and Pentium III chipsets. For cryptographic applications, it is important to use as much pseudo-random data as reasonably possible when seeding a RNG.

One simple way of obtaining reasonably high-quality seed data is to use Microsoft's Crypto API, which is installed on most 32-bit Windows systems. The following sample code demonstrates how the module can be used to generate a random seed:

```
/* Windows program to demonstrate obtaining seed and using it with
 * MOVEitCrypto.  Link with Advapi32.lib and MOVEitCrypto.lib.  */
#include <windows.h>
#include <wincrypt.h>
#include <stdio.h>
#include "moveitcrypto.h"

int main(int argc, char *argv[])
{
   MC_CONTEXT context;
   MC_ERROR mcerror;
   unsigned char seed[256], ranbytes[10];
   int j;

   /* Use the MS Crypto API to generate a seed. */
   HCRYPTPROV hProv = 0;
   CryptAcquireContext(&hProv, 0, 0, PROV_RSA_FULL, CRYPT_VERIFYCONTEXT);
   CryptGenRandom(hProv, sizeof(seed), seed);
   /* Now the seed is in "seed".  Release the MS Crypto API context. */
   CryptReleaseContext(hProv, 0);

   /* Set the license key.  (Don't use this dummy key in your code.) */
   mcerror=MCSetLicenseKey("12345-12345-12345");
   /* Initialize the RNG. */
   mcerror=MCRNGInit(&context);
   /* Feed the seed to MOVEit Crypto, then zero the local copy. */
   mcerror=MCRNGUpdate(&context, seed, sizeof(seed));
   memset(seed, 0, sizeof(seed));
   /* Now it's OK to call the RNG. */
   mcerror=MCRNGGenerate(&context, sizeof(ranbytes), ranbytes);
   if(mcerror) printf("MCRNGGenerate returned %d\n", mcerror);
   /* Insert code here to use ranbytes. */
   printf("Got random bytes:");
   for(j=0; j<sizeof(ranbytes); j++) printf(" %-2.2x", ranbytes[j]);
   /* Clear the context. */
   mcerror=MCRNGFinal(&context);

   return 0;
}
```

On a Linux or FreeBSD system, a program can obtain reasonably high-quality seed data by reading the file /dev/random.

Protecting Keys and Data

Although MOVEit Crypto is careful to zeroize its own copies of sensitive information, it has no control over keys and data managed by the calling application. For maximum security, be sure to zeroize all copies of key material and plaintext data when they are no longer needed by the application. This applies to memory, disk, and other forms of storage. Be aware that simply deleting files or returning allocated memory buffers to the system will not destroy the data.

MOVEit Crypto Context

The layout of MC_CONTEXT may change from one release of the module to another. For this reason, and for security reasons, the context must not be saved between invocations of a program.

Crypto Officer Guidance

With respect to the MOVEit Crypto module, the responsibilities that apply to a cryptographic officer that do not apply to a user are:

- Installing or uninstalling the module.
- Configuring the operating system.

Since MOVEit Crypto does not, by itself, persistently store any keys or sensitive information on the system, no special action need be taken to administratively protect such information. However, be aware that applications using the module may store sensitive information on their own.

Guidance for Windows

MOVEit Crypto is installed via a typical installation program. You must be logged in as an Administrative user to install the module, and you must have write access to the Windows system directory, as well as a second selectable directory for the documentation and other support files.

To uninstall the module, use the Control Panel Add/Remove Programs applet.

It is recommended that the cryptographic officer configure the operating system to minimize the likelihood that an attacker could obtain sensitive information from system paging or dump files. This can be done by setting certain registry values:

- Set HKEY_LOCAL_MACHINE \ SYSTEM \ CurrentControlSet \ Control \ CrashControl \ CrashDumpEnabled to 0. This is a REG_DWORD value.
- Set HKEY_LOCAL_MACHINE \ SYSTEM \ CurrentControlSet \ Control \ Session Manager \ Memory Management \ ClearPageFileAtShutdown to 1. This is a REG_DWORD value.

Guidance for Linux

Install / Uninstall

MOVEit Crypto is installed via Red Hat Package manager, rpm. You must be logged in as a root user to install the module. To install the module, use the command

```
rpm -i moveitcrypto-1.1-1.i386.rpm
```

To uninstall the module, login as a root user and use the command

```
rpm -e moveitcrypto
```

Preventing core Files

It is recommended that the cryptographic officer ensure that users are not configured to produce "core dump" files when their application faults, unless this capability is necessary during development and testing. The resulting core dump files can contain sensitive information, including keys and data.

Many systems are configured to not create core files by default, but the cryptographic operator may want to modify the user's configuration to ensure this. The method of preventing "core" files from being produced depends on which shell is being used by the user in question.

For users running the bash shell, which is the default under Red Hat Linux, the cryptographic officer can add the line

```
ulimit -c 0
```

to the end of the .bashrc file for the user in question to prevent creation of core dump files.

For users running csh or tsch, the cryptographic officer can add the line

```
limit coredumpsize 0
```

to the end of the .cshrc file for the user in question to prevent creation of core dump files.

Other Linux Security Measures

The cryptographic officer should ensure that the file /etc/ld.so.preload does not exist (by default it does not exist). This will help prevent other libraries from usurping the functions used by MOVEit Crypto.

Guidance for FreeBSD

Install / Uninstall

To install MOVEit Crypto under FreeBSD, login as root, download moveitcrypto-1.1.dist.tar.gz into some directory (/tmp in this example) and type

```
cd /
tar xfzv /tmp/moveitcrypto-1.1.dist.tar.gz
/usr/local/bin/moveitcrypto-install
```

To uninstall, login as root and type

```
/usr/local/bin/moveitcrypto-remove
```

Preventing core Files

See the corresponding section for Linux.

Other FreeBSD Security Measures

See the corresponding section for Linux.

Miscellaneous

Mitigation of Specific Attacks

MOVEit Crypto is not designed to mitigate specific attacks.

Tabular Summaries

As required by FIPS 140-2 Derived Test Requirements, here are tables summarizing certain aspects of the security policy:

Role	Type of Authentication	Authentication Data
User	None	N/A

Crypto Officer	None	N/A

Roles and Required Identification and Authentication

Authentication Mechanism	Strength of Mechanism
User	None
Crypto Officer	None

Strengths of Authentication Mechanisms

Role	Authorized Services
User	All
Crypto Officer	All

Services Authorized for Roles

Service	Cryptographic Keys and Critical Security Parameters	Type(s) of Access (e.g., Read, Write, Execute)
MCEncryptInit, MCDecryptInit	Encryption keys	W
MCEncrypt, MCDecrypt	Encryption keys	E
MCEncryptFinal, MCDecryptFinal	Encryption keys	WE (W to zeroize only)
MCKeyedHashInit	HMAC keys	W
MCKeyedHashUpdate	HMAC keys	E
MCKeyedHashFinal	HMAC keys	WE (W to zeroize only)
MCRNGInit	RNG seeds	W (to zeroize only)
MCRNGUpdate	RNG seeds	W
MCRNGGenerate	RNG seeds	E
MCRNGFinal	RNG seeds	W (to zeroize only)
MCZeroizeContext	Encryption keys, HMAC keys, and RNG seeds	W (to zeroize only)
MCSelfTest	Built-in HMAC key for software/firmware test	E
All other	None	N/A

Access Rights within Services

Physical Security Mechanisms	Recommended Frequency of Inspection / Test	Inspection / Test Guidance Details
None	N/A	N/A

Inspection/Testing of Physical Security Mechanisms

Other Attacks	Mitigation Mechanism	Specific Limitations
None	N/A	N/A

Mitigation of Other Attacks