SafeAPI - v1.20
User Guide
January 27, 2005

SafeLogic S.A.R.L, € 30 489 in capital
Trade Register: Paris B 422 461 848
27-29, rue Raffet - 75016 Paris
Tel: (33) (0)1 45 72 25 15 - Fax: (33) (0)1 45 72 14 06 - Web: www.safelogic.com

Contents

1 INTRODUCTION

1.1 Contents of This Guide

This guide is the user documentation for the SafeAPI Cryptography API for Windows NT/2000/2003/XP and Unix.

For Windows NT/2000/2003/XP, it discusses how to use this library and illustrates the explanation using Visual Basic:

- Parameter settings for Visual Basic 6.0 projects
- Description of the API: functions, arguments, return values, and error codes.

1.2 Intended Audience

This guide is written for Java developers (or developers using Visual Basic, C/C++, Delphi, PowerBuilder, etc.) who want to incorporate calls to the SafeAPI cryptography API. The SafeAPI cryptography API is designed to simplify the development of applications that use hashing, signature, and symmetric or asymmetric encryption functions.

A basic knowledge of cryptography concepts is needed to understand this document.

1.3 Environment

SafeAPI was tested in the following environments:

Environment	**Versions**
Windows NT	• Windows NT Server V 4.0 SP 4 or higher. • Windows NT Workstation V 4.0 SP 4 or higher.
Windows 2000/XP	• Windows 2000 Server SP 2 or later • Windows 2000 Professional SP 2 or higher. • Windows XP Professional SP 1 or higher.
Unix	• Sun Solaris 7.x/8.x/9.x (SPARC and Intel). • Linux (Intel) RedHat 6.1/7.1/8.0/9.0. • Other Unix versions which support Java.
JVM (Java Virtual Machine)	• Sun Microsystems JDK 1.4.0, JDK 1.4.1 or JDK 1.4.2.

Support for this API is provided for all these Java environments and any newer versions.

2 Installing SafeAPI

2.1 Installing the JDK 1.4 from Sun Microsystems.

SafeJDBC requires the installation of the JDK 1.4 from Sun Microsystems.

The installation of the JDK 1.4 is described on Sun Microsystems' website at the following address: http://java.sun.com/j2se. This page also contains downloads for other environments: Sun Solaris, Itanium, etc.

2.2 Installing the SafeAPI binary files

Download `safeapi_v1.20.zip` file at:
http://www.safelogic.com/safeapi/download/safeapi_v1.20.zip

2.2.1 Standard installation

We suggest that the following installation directories be used by default:

Windows NT/2000/XP	Unzip and copy the **`\safelogic`** directory to **`c:\`**
Unix/Linux	Create a user named safelogic, unzip **and** copy the **`\safelogic`** directory in the WinZip file into **`/home`**

This is the resulting Windows tree:

Windows Directory	Comments
`c:\safelogic\lib`	Java common libraries.
`c:\safelogic\safeapi`	SafeAPI base directory.
`c:\safelogic\safeapi\conf`	SafeAPI scripts for classpath.
`c:\safelogic\safeapi\doc`	User Documentation.
`c:\safelogic\safeapi\examples`	Sample Java programs.
`c:\safelogic\safeapi\javadoc`	Javadoc.
`c:\safelogic\safeapi\keys`	Keys & license directory.
`c:\safelogic\safeapi\lib`	Java SafeAPI libraries. (Includes DLL).

This is the resulting Unix/Linux tree:

Unix/Linux directory	Comments
`/home/safelogic/lib`	Java common libraries.
`/home/safelogic/safeapi`	SafeAPI base directory.
`/home/safelogic/safeapi/conf`	SafeAPI scripts for classpath.
`/home/safelogic/safeapi/doc`	User Documentation.
`/home/safelogic/safeapi/examples`	Sample Java programs.
`/home/safelogic/safeapi/javadoc`	Javadoc.
`/home/safelogic/safeapi/keys`	Keys & license directory.
`/home/safelogic/safeapi/lib`	Java SafeAPI libraries. (Includes DLL).

2.2.2 Customized installation

SafeAPI may be installed in any directory.

The only constraint is to update the CLASSPATH (defined below) with the corresponding files and directories.

2.3 LICENSE FILE

SafeAPI uses a license file called `safeapi_license.txt.` The lines contains :

- The keyword “safeapi »
- A license code for the type of product.
- The server names list (HOSTNAME), separated by “;”
- A product evaluation end date or 99999999 for full versions.
- Your email address.
- A hexadecimal string corresponding to the signature for the above elements with a private RSA key.

By default, `safeapi_license.txt` is installed in the `keys` directory. (`c:\safelogic\safeapi\keys` or `/home/safelogic/safeapi/keys`)

If you choose a customized installation, please follow these constraints:

- `safeapi_license.txt` must be installed in a directory defined in the classpath.

- `safeapi_license.txt` must be installed in the same directory as `license@safelogic.com_1_RSA.pkf`. (The default file installation is in the `keys` directory).

2.4 Updating the Java CLASSPATH

The `CLASSPATH` Java environment variable must include:

1. Access path to the two libraries `cryptix.jar` and `safeapix.jar`.
2. Access path to the `keys` directory which contains `safeapi_license.txt` and `license@safelogic.com_1_RSA.pkf`.
3. Access path to the `examples` directory which contains Java samples.

2.4.1 Windows NT/2000/XP Classpath

For the standard installation, add the following string to the CLASSPATH:

```
c:\safelogic\lib\cryptix.jar;c:\safelogic\safeapi\lib\safeapix.jar;
c:\safelogic\safeapi\keys;c:\safelogic\safeapi\examples\java;
```

2.4.2 Unix/Linux classpath

For the standard installation, add the following string to the CLASSPATH:

```
/home/safelogic/lib/cryptix.jar;/home/safelogic/safeapi/lib/safeapix.jar;
/home/safelogic/safeapi/keys;/home/safelogic/safeapi/examples/java;
```

2.5 DLL installation (Windows)

- Copy the `safejnidll.dll` & `safeapix.dll` from `c:\safelogic\safeapi\lib` into a "system" folder on the hard drive.
- Example: `c:\WINDOWS\system32`.There are no constraints on where to put the two DLL.

Registering safeapix.dll

If the DLL is installed in system32, it must be registered:

- Open an MS-DOS session on your workstation.
- `CD C:\WINDOWS\System32.`
- Run the `regsvr32` command on the DLL. Example:

```
C:\WINDOWS\System32>regsvr32 safeapix.dll
```

2.6 VERIFYING AND TESTING THE INSTALLATION

Open an MS-DOS or Bash session, and run the sample Java program **`TestHash`** with the string `"abc"` as the parameter:

```
java com.safeapi.test.TestHash "abc"
```

The program prints to the screen the MD5 and SHA-1 values for "abc":

```
MD5: 900150983CD24FB0D6963F7D28E17F72
SHA-1: A9993E364706816ABA3E25717850C26C9CD0D89D
```

2.7 Javadoc

SafeAPI Javadoc is in the **`/safelogic/safeapi/javadoc`** `directory` in `safeapi_v1.20.zip`.

2.8 Javadoc on line

SafeAPI Javadoc is browsable online at:
http://www.safelogic.com/safeapi/v1.20/javadoc.

3 Configuring the Technical Environment in Windows

3.1 General Information

SafeAPI is delivered in the form of functions that can be called from any development environment supporting the use of DLL.

Refer to the technical documentation for your development environment for instructions on how to call DLL functions.

We have selected a configuration that uses Microsoft Visual Basic 6.0 to illustrate the general approach.

3.2 Example: Microsoft Visual Basic 6.0

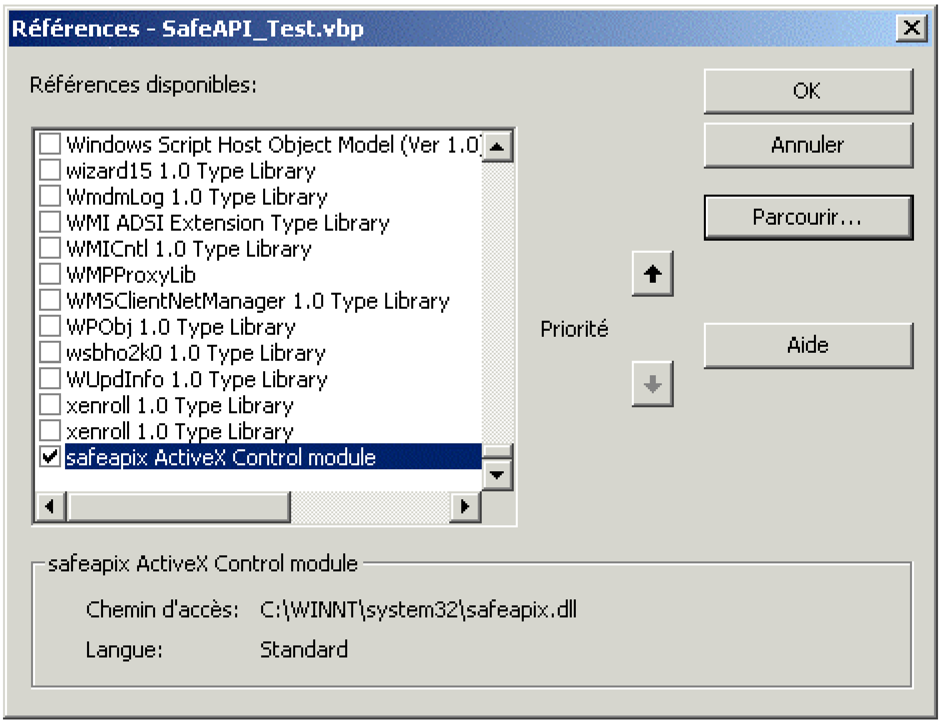
In the Visual Basic environment, the DLL components must be "imported" in order to use the cryptography API. This is done by adding the **safeapix.dll** to the project references.

Here is a brief summary of the steps to follow:

- Create a new VB Project.
- **Project** menu - **References** sub-menu. In the list of available References, **safeapix** should appear. Click the checkbox, then click the OK button.

If the DLL does not appear in the list of **Available references**, use the **Browse** button to locate it (see image below).

The end result of this operation should resemble this screen shot:



Références disponibles:

- ☐ Windows Script Host Object Model (Ver 1.0)
- ☐ wizard15 1.0 Type Library
- ☐ WmdmLog 1.0 Type Library
- ☐ WMI ADSI Extension Type Library
- ☐ WMICntl 1.0 Type Library
- ☐ WMPProxyLib
- ☐ WMSClientNetManager 1.0 Type Library
- ☐ WPObj 1.0 Type Library
- ☐ wsbho2k0 1.0 Type Library
- ☐ WUpdInfo 1.0 Type Library
- ☐ xenroll 1.0 Type Library
- ☐ xenroll 1.0 Type Library
- ☑ safeapix ActiveX Control module

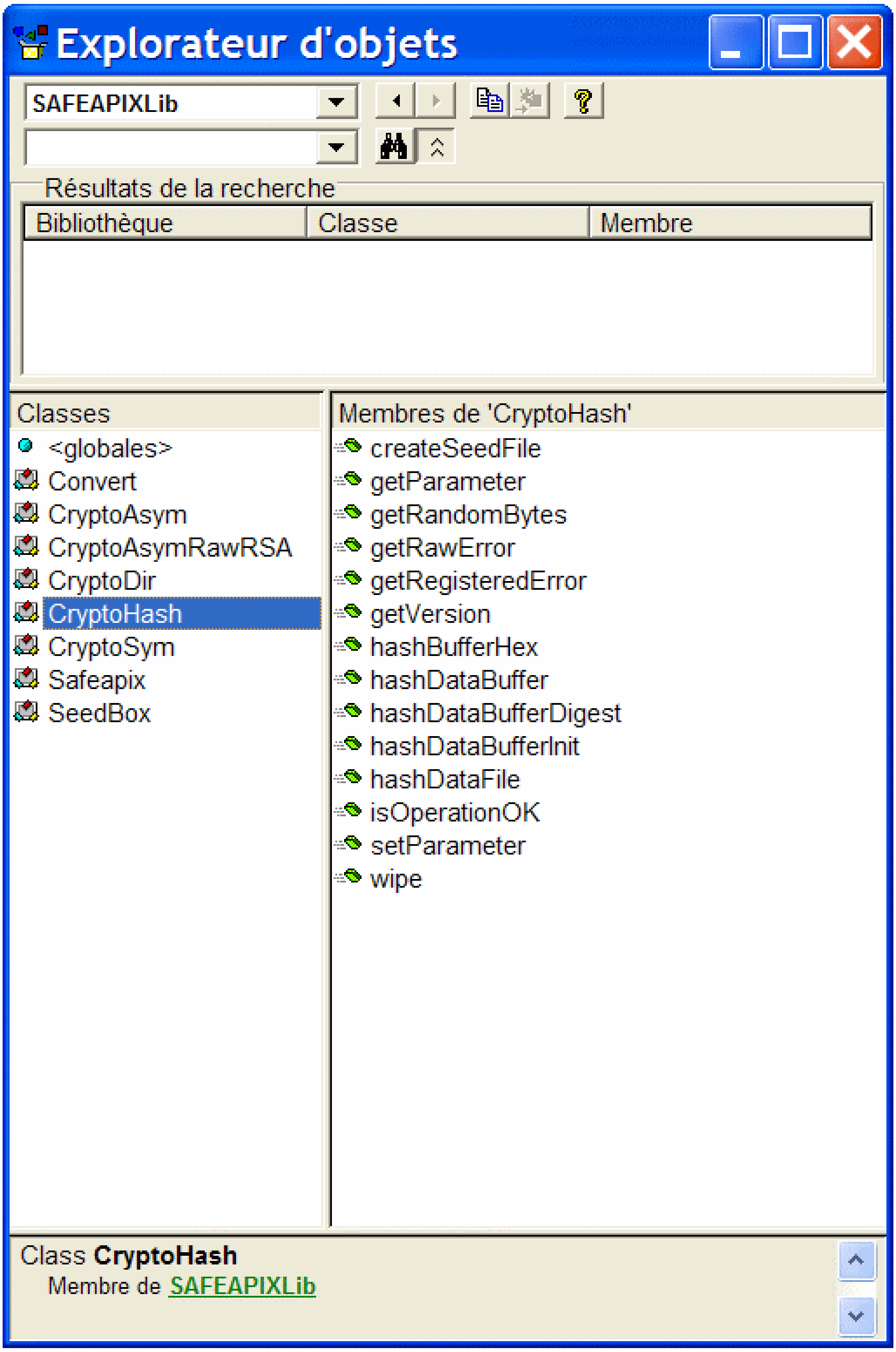
safeapix ActiveX Control module

Chemin d'accès: C:\WINNT\system32\safeapix.dll

Langue: Standard

You can verify that a component of the DLL is available by using the Object Browser. This is accessed through an icon in the toolbar, the **F2** shortcut key, or the **View** menu.

If the DLL is correctly integrated with the project, you should be able to select **safeapix** in the list of libraries:



Classes:
- <globales>
- Convert
- CryptoAsym
- CryptoAsymRawRSA
- CryptoDir
- CryptoHash
- CryptoSym
- Safeapix
- SeedBox

Membres de 'CryptoHash':
- createSeedFile
- getParameter
- getRandomBytes
- getRawError
- getRegisteredError
- getVersion
- hashBufferHex
- hashDataBuffer
- hashDataBufferDigest
- hashDataBufferInit
- hashDataFile
- isOperationOK
- setParameter
- wipe

Class **CryptoHash**
Membre de SAFEAPIXLib

4 USING SAFEAPI

4.1 Classes

SafeAPI is a set of **classes**, as defined by **object oriented** terminology. A class must be **instanced** prior to any method call. A class is a group of **methods** (or **functions**).

The usage is the same for any calling language whether or not the DLL is used: Visual Basic, C/C++, Delphi, PowerBuilder, etc. Note that SafeAPI does **not** require the calling program to be object oriented.

For example, the `CryptoHash` class is dedicated to hash computations. The `hashDataFile` computes the hash value of any file. The result is returned as a byte array.

The following is an example of MD5 hash computation with Java, C++ and Visual Basic:

Java

```
    // Loads an instance of CryptoHash
    CryptoHash cryHash = new CryptoHash();

    // Compute the Hash value of safeapix.dll with MD5 algorithm
    byte[] hash = cryHash.hashDataFile("MD5",
                         "c:\\windows\\system32\\safeapix.dll");
    // Etc...
```

Visual C++ 6.0

```
    // Loads an instance of CryptoHash
    ICryptoHash cryHash;

    if (!cryHash.CreateDispatch("safeapix.CryptoHash"))
        AfxMessageBox("Error! SafeAPI CryptoHash not loaded!" ,0 ,0);

    // Compute the Hash value of safeapix.dll with MD5 algorithm
    COleVariant hash = cryHash.hashDataFile("MD5",
                              "c:\\windows\\system32\\safeapix.dll");
    // Etc...
```

Visual Basic 6.0

```
    ' Loads an instance of CryptoHash
    Dim cryHash As New CryptoHash

    ' Compute the Hash value of safeapix.dll with MD5 algorithm
    Dim hash As Variant
    hash = cryHash.hashDataFile("MD5",
                  "c:\\windows\\system32\\safeapix.dll")
    ' Etc...
```

4.2 List of SafeAPI classes

SafeAPI classes in alphabetic orders:

Classes	Methods
Convert	Conversion tools and methods
CryptoAsym	Asymmetric Cryptography API methods
CryptoAsymRawRSA	Pure/raw RSA Asymmetric Cryptography API methods
CryptoCommon	Defines methods for all SafeAPI sister CryptoXxx classes; must not be used directly and is instanced by sister CryptoXxx classes: *CryptoAsym, CryptoAsymRawRSA, CryptoDir, CryptoHash, CryptoSym.*
CryptoDir	Directory encryption/decryption API methods.
CryptoHash	Hash methods for buffers and files.
CryptoSym	Symmetric Cryptography API methods.
Parms	This class defines the parameters that API users may get and modify using `CryptoCommon.getParameter` and `CryptoCommon.setParameter`.
SeedBox	Seed generation visual methods
Status	Contains all the error & return codes as Strings.

4.3 List of SafeAPI classes per License Type

Access to the classes is granted depending on the SafeAPI License type:

SafeAPI License Type	Accessible Classes
Evaluation License	All classes (with time limit).
Hash License	Convert, **CryptoHash,** Status
Symmetric Cryptography License	*Hash License Classes,* **CryptoSym, SeedBox**
Asymmetric Cryptography License	*Symmetric Cryptography License,* **CryptoAsym, CryptoAsymRawRSA**
Directory Encryption License	All classes: Asymmetric Cryptography License, **CryptoDir**

5 CLASS REFERENCE MANUEL

5.1 Class Convert: String Conversion Methods

This is a class of type conversion utilities provided to simplify using the API:

- Converts bytes to a string.
- Converts bytes to a hexadecimal value.

Method	**bytesToHexString**		
Description	Converts a binary value into hexadecimal format.		
Arguments	byte [ ]	bBuffer	Buffer containing bytes.
Return value	String		Buffer converted into a string of hexadecimal characters.

Method	**bytesToString**		
Description	Converts a buffer of bytes into a string.		
Arguments	byte [ ]	bBuffer	Buffer containing bytes.
Return value	String		Buffer converted into a string of characters.

Method	**hexStringToBytes**		
Description	Converts a string *containing a hexadecimal representation* into a byte [ ].		
Arguments	String	sHexChain	Hexadecimal representation of the string.
Return value	byte [ ]		Hexadecimal string converted into bytes.

Method	**stringToBytes**		
Description	Converts a string into a buffer containing bytes.		
Arguments	String	sChain	String of characters.
Return value	byte [ ]		The string converted into bytes.

5.2 Class SeedBox: Visual Methods for Generating Seeds

A class for randomly generating seeds. The user is asked to enter data into a dialog box.

The seed value is created in order to initialize a pseudo-random number generator. Seed values are widely used to generate symmetric or asymmetric keys whose values cannot be detected by an attacker.

Method	**seedDialog**		
Description	Generates a random seed of the desired length by asking the user to enter data into a dialog box. Returns the seed.		
Arguments	String	sTitle	Title of the dialog box [optional]
	String	sCaption	Caption for the dialog box [optional]
	int	lSeedSize	Length of the seed (in bits)
Return value	byte [ ]		The seed is returned as a series of bytes

Method	**getSeedValue**	
Description	Returns the value of the generated seed.	
Arguments	None	
Return value	byte [ ]	The seed is returned as a series of bytes

Method	**seedCanceledByUser**	
Description	Indicates whether the user canceled the seed generation (by closing the dialog box).	
Arguments	None	
Return value	Boolean	True: the user closed the box before normal completion False: the seed generation completed normally

An example of using this class in VB is the method **bt_GenSeed_Click** found in the implementation example. This method is found in the **Form_GenKeyTest.frm** file, which corresponds to the GenKeyTest window.

5.3 Class CryptoCommon: common methods of CryptoXxx classes

CryptoCommon cannot be *instanced and is hidden in the DLL*.

It contains common methods for the following classes:

- **CryptoAsym,**
- **CryptoAsymRawRSA,**
- **CryptoDir,**
- **CryptoHash,**
- **CryptoSym.**

5.3.1 CryptoCommon : Parameters setting API

These methods allow you to specify the system environment where the cryptography functions will be used, directly from your development environment.

This type of implementation eliminates the use of static parameter settings in the DLL and/or the ".ini" files, a source of errors and attacks.

The parameters that can be set dynamically concern:

- the directory for storing keys.
- the error report file (dump file).

Method	**setParameter**		
Description	Used to change the value of an API parameter.		
Arguments	String	ParamName	Name of the parameter to be changed (see list below)
	String	Value	New value for the indicated parameter.
Return value	Boolean		True: the change was made False: the change was not made

Method	**getParameter**		
Description	Used to retrieve the current value of an API parameter.		
Arguments	String	ParamName	Parameter name (see list below)
Return value	String	The value of the parameter called ParamName.	

5.3.2 List of Modifiable Parameters

Parameter	Role
DEFAULT_SIGN_ALGO	Default signature algorithm used.
DEFAULT_SYM_ALGO	Default symmetric algorithm used in asymmetric cryptography and in combined asymmetric/symmetric cryptography.
DUMP_ENABLED	**True** or False: Indicates whether the API should create a dump file detailing the unknown errors of error code **CRYPTO_UNKNOWN_ERROR.**
DUMP_FILEPATH	Complete name of the dump file (written in append mode).
KEY_DIRECTORY	Directory for storing keys.
MIN_PASSPHRASE_SIZE	Minimum size allowed for a passphrase (number of characters).
RAND_SEED_DIR	Directory for storing seed files.

5.3.3 List of the Parameter Default Values

Parameter	Default value
DEFAULT_SIGN_ALGO	`RSA`
DEFAULT_SYM_ALGO	`Blowfish`
DUMP_ENABLED	`False`
DUMP_FILEPATH	`user.home directory`
RAND_SEED_DIR	`Directory that contains safeapi_license.txt. (keys).`
MIN_PASSPHRASE_SIZE	`8`
KEY_DIRECTORY	`Directory that contains safeapi_license.txt. (keys).`

5.3.4 Error Detection Methods

The general approach for diagnosing an error is as follows:

- 1) Verify that the operation just performed has completed properly.
- 2) If not, request the type of error.
- 3) A detailed report is available if the error type does not provide enough information. This may be the case for system errors.

This procedure is followed by using three methods, one for each of the three tasks described:

Method	**isOperationOK**	
Description	Indicates whether or not the call to the last cryptography method finished correctly.	
Arguments	None.	
Return value	Boolean	True: the last method executed successfully False: the last method failed

Method	**getRegisteredError**	
Description	Indicates the reason why the call to the last cryptography method failed.	
Arguments	None	
Return value	String	Error message (diagnostic error code)

Method	**getRawError**	
Description	Returns detailed information on the last error, as a string.	
Arguments	None.	
Return value	String	Detailed error message.

The list of error messages returned by this function is provided in a table found in the appendix (**Appendix A: Error Code Table by Method).**

The table also lists the possible errors for each cryptography method (values of **getRegisteredError** and **getRawError).**

5.4 Methods for Creating Seeds or Random Values

5.4.1 Initial Creation of a File of Random Numbers (seed file)

Method	createSeedFile		
Description	Creates (or replaces) the file used to seed the random number generator. Uses a random number generator dialog box to prompt the user.		
Arguments	String	Title	The title for the seed dialog box
Return Value	String	Caption	The caption for the seed dialog box

Notes:

- A seed file must be created before using the symmetric and asymmetric encryption functions.

- The *random* data are passed as input to a *pseudo-random* number generator that is based on the ANSI X9.17 standard and implemented by the **getRandomBytes()** method.

- Call **setParameter()** with "RAND_SEED_DIR" to specify a directory for storing the seed file.

5.4.2 Dynamic Seed Generation

Method	getRandomBytes		
Description	Generates a seed or pseudo-random value of 24 bytes.		
Argument	String	sMsg	Initialization message. A string containing any characters
Return value	byte [ ]	Hash value for the data buffer (series of bits).	

Notes:

- A seed file must be created using **createSeedFile()** before calling **getRandomBytes().**

- The seed values are passed as data to the symmetric or asymmetric key generation functions (**bSeed** parameter).

5.5 Other Methods

Method	**getVersion**	
Description	Gets the SafeAPI Version	
Arguments	None.	
Return value	String	SafeAPI Version information.

Method	**wipe**		
Description	Destroys the specified file in a secure manner, by writing random data over the file contents in several passes.		
Arguments	String	FilePath	Complete name of the file to be wiped.
	int	Level	Security level, from 1 to 3 (8 to 20 passes)

Note:

- **Wipe** provides a way to delete a file so it cannot be retrieved.
- Attention: a file erased by **wipe** is totally unrecoverable.

5.6 Class CryptoHash: Hash Methods

CryptoHash contains:

- Hash methods for unlimited size buffers.
- Hash methods for files.
- Hash methods for small size buffers.

5.6.1 Hashing a Buffer

Method	**hashDataBufferInit**		
Description	Initializes the tool and the buffer to be hashed.		
Arguments	String	Algorithm	Hash algorithm: • `"MD5"` • `"SHA-1"`

Method	**hashDataBuffer**		
Description	Adds to the buffer the bytes passed as an argument. The buffer is not hashed until hashDataBufferDigest is called. This method should only be used after the buffer has been initialized.		
Arguments	byte [ ]	bSubBuffer	Bytes to be added to the hash buffer

Method	**hashDataBufferDigest**	
Description	Calculates and returns the hash value for the buffer, which was first initialized and filled with data by calling the appropriate methods.	
Arguments	None.	
Return value	byte [ ]	Hash value for the data buffer (series of bits)

Notes:

The hash value for the buffer/string is done by accumulation, so hashing can be done on strings of unlimited size. The steps must be performed in this sequence, however:

- 1) Initialization: **hashDataBufferInit.**
- 2) Add data to the hash buffer: **hashDataBuffer.**
- 3) Calculate the hash value: **hashDataBufferDigest.**

5.6.2 Hashing a File

Method	**hashDataFile**		
Description	Calculates and returns the hash value associated with the specified file, using the algorithm specified as an argument.		
Arguments	String	sAlgorithm	Hash algorithm: • `"MD5"` • `"SHA-1"`
	String	sFilePath	Complete file name
Return value	byte [ ]	Hash value for the data buffer (series of bits).	

Notes:

- The term **complete name** means the complete pathname for the file, including the directories and disk drive.
  - Windows NT/2000 example: **`c:\safelogic\lib\cryptix.jar`**
  - Unix example: **`/home/safelogic/lib/cryptix.jar`**

5.6.3 Hashing a small size Buffer

Method	**hashBufferHex**		
Description	Calculates and returns the hash value associated with the specified buffer, using the algorithm specified as an argument		
Arguments	String	sAlgorithm	Hash algorithm: • `"MD5"` • `"SHA-1"`
	String	bBuffer	Buffer to hash
Return value	String	Hash value for the data buffer (hexadecimal String).	

Notes:

- Allows hashing with a single operation.

- The return value is a hexadecimal String. This allows the result to be displayed on the screen and/or to be stored directly without prior conversion.

5.7 Class CryptoSym: Symmetric Encryption Methods

5.7.1 Generating a Secret Key

Method	**genSecretKey**		
Description	Generates a secret key of the desired length (if valid) using the specified algorithm, based on the seed provided. Returns the key.		
Arguments	char [ ]	caPassphrase	Passphrase protecting the key.
	String	sAlgorithm	Symmetric encryption algorithm. (See "Table of Symmetric Encryption Algorithms")
	int	Length	Key length (in bits)
	byte [ ]	bSeed	Value of the seed.
	String	sEmail	Reference email for the key.
	int	lKeyIndex	Index number for the key (optional).
Return value	None.		

Notes:

The generated secret key is stored in a file whose name is formed by concatenating the following objects in the order listed, separated by the character "_":

- The reference email.
- The index passed to **genSecretKey.**
- The name of the algorithm selected: Blowfish, CAST-128, IDEA.
- The extension ".skf ".

The "**Key ID"** for the secret key is the lower-case generic name for the file, without any directories or extensions.

Key ID examples:
- **user@domain.com_1_ cast-128**
- **user@domain.com_2_ blowfish**

Remember:
The directory where this file is stored is defined by calling the **setParameter()** method and passing it the **KEY_DIRECTORY** parameter.

Before the secret key is saved on the hard drive, it is encrypted with:

- The algorithm selected by the user in the API: Blowfish, CAST-128, IDEA.
- CFB mode.
- A 128-bit key that is the MD5 hash value for the passphrase parameter passed to **genSecretKey.**

The secret key is therefore stored in a highly secure form on the hard drive, and the passphrase is not stored.

5.7.2 Encrypting a Buffer

Method	encryptBuffer		
Description	Encrypts the buffer passed as an argument, using the key associated with the specified passphrase.		
Arguments	String	sKey_ID	Key ID for the secret key.
	char [ ]	caPassphrase	Passphrase protecting the key sKey_ID.
	byte [ ]	bBuffer	Buffer to be encrypted.
Return value	byte [ ]		Encrypted buffer, preceded by the IV (Initialization Vector) generated internally by the method.

Reminder:

The **Key_ID** for the secret key is an identifier in the format:

```
<email>_<index>_<algorithm>
```

where:
`e_mail:` POP account associated with the key, such as **user@domain.com.**
`index:` Numeric index.
`algorithm:` Algorithm used. Must be either Blowfish, CAST-128, or IDEA.

Method	decryptBuffer		
Description	Decrypts the buffer passed as an argument, using the key associated with the specified passphrase.		
Arguments	String	sKey_ID	Key_ID for the secret key.
	char [ ]	caPassphrase	Passphrase protecting the key sKey_ID.
	byte [ ]	bBuffer	Buffer *encrypted* with ***encryptBuffer*** and now to be decrypted.
Return value	byte [ ]		Decrypted buffer.

Notes:

This method must be used *jointly* with **encryptBuffer** to encrypt and decrypt buffers. Here is the process:

- **encryptBuffer:** encrypts a buffer and returns the encrypted buffer preceded by 64 bits of Initialization Vector.
- **decryptBuffer:** decrypts a buffer *encrypted* using ***encryptBuffer***, so the first 64 bits contain an IV.

The IV represents the first 64 bits of the returned buffer. This concatenation is completely transparent during normal encryption followed by later decryption of the buffer.

5.7.3 Encrypting Buffers: Overloaded Methods With Key and IV Arguments

The SafeAPI methods for symmetric encryption/decryption of buffers are overloaded to allow experienced users to manage the parameters of a symmetric encryption directly:

- Key value in binary format.
- IV (Initialization Vector) value in binary format.

Method	**encryptBuffer**		
Description	Encrypts the buffer passed as an argument, using the key and the IV specified in byte form.		
Arguments	String	sAlgorithm	Encryption algorithm (See "Table of Symmetric Encryption Algorithms")
	byte [ ]	bKey	128-bit key in binary format.
	byte [ ]	bIV	The 64-bit IV (8 bytes) to be used.
	byte [ ]	bBuffer	Buffer to be encrypted.
Return value	byte [ ]		Encrypted buffer

Method	**decryptBuffer**		
Description	Decrypts the buffer passed as an argument, using the key and the IV specified in byte form.		
Arguments	String	sAlgorithm	Encryption algorithm (See "Table of Symmetric Encryption Algorithms")
	byte [ ]	bKey	128-bit key in binary format.
	byte [ ]	bIV	64-bit IV (8 bytes) used to encrypt the buffer.
	byte [ ]	bBuffer	Buffer to be decrypted.
Return value	byte [ ]		Decrypted buffer.

Notes:

- The three algorithms are always used in CFB mode (Cipher Feedback).

- These overloaded methods are also used to test the operation of SafeAPI and VB programs in comparison to test sets. (See the "IMPLEMENTATION/CRYPTOGRAPHY FAQ" for more details.)

5.7.4 Encrypting Files

Method	**encryptFile**		
Description	Encrypts the file whose complete name (including drive and directories) is passed as an argument, using the key associated with the specified passphrase.		
Arguments	String	sKey_ID	Key_ID for the key.
	char [ ]	caPassphrase	Passphrase protecting the key sKey_ID.
	String	sInputPath	Complete name of the file to be encrypted.
	String	sOutputPath	Complete name of the encrypted file.

Notes:

- **sInputPath** and **sOutputPath** must be different.
- Reminder: The term **complete name** means the complete pathname of a file, including the drive and directories.
  - Windows NT/2000 example: **`c:\safelogic\lib\cryptix.jar`**
  - Unix example: **`/home/safelogic/lib/cryptix.jar`**

Method	**decryptFile**		
Description	Decrypts the file whose complete name is passed as an argument, using the key associated with the specified passphrase.		
Arguments	String	sKey_ID	Key_ID for the key.
	char [ ]	caPassphrase	Passphrase protecting the sKey_ID key.
	String	sInputPath	Complete name of the file to be decrypted
	String	sOutputPath	Complete name of the decrypted file

Notes:

- **sInputPath** and **sOutputPath** must be different.

5.7.5 Table of Symmetric Encryption Algorithms

Algorithm	Comments
`Blowfish`	CFB (Cipher Feedback) mode.
`CAST-128`	CFB mode.
`IDEA`[1]	CFB mode.

Note: the algorithm name must appear exactly as shown above.

---

[1] IDEA is patented by ASCOMM and requires a special license from SafeLogic to be used.

5.8 Class CryptoAsym: Asymmetric Encryption and Signature Methods

5.8.1 Generating a Pair of RSA Keys

Method	genKeyPair		
Description	Generates a pair of RSA keys of the desired length (if valid) by applying the specified algorithm to the provided seed. Returns the pair of keys.		
Arguments	String	sAlgorithm	Asymmetric encryption algorithm: • "RSA".
	char [ ]	caPassphrase	Passphrase protecting the key.
	int	Length	Length of the key (in bits): must be a multiple of 512 that is less than or equal to 4096.
	byte [ ]	bSeed	Seed value.
	String	sEmail	Reference email for the key.
	int	lKeyIndex	Index number for the key (optional).
Return value			

Notes:

The file names for the files containing the two keys (public & private) are built from:

- The reference email.
- The index passed to **genKeyPair.**
- The string "**RSA**".
- The extension "**.skf**" for the private key.
- The extension "**.pkf**" for the public key.

The "**Key ID**" for the pair of keys is the lower-case generic name for the file, without directories or extensions.

Key ID examples:
- **user@domain.com_1_ rsa**
- **user@domain.com_2_ rsa**

Before the private key is saved on the hard drive, it is encrypted using:
- The Blowfish symmetric algorithm.
- CFB mode.
- A 128-bit key which is the MD5 hash value for the passphrase parameter passed to **genKeyPair.**

The private key is therefore encrypted and stored in a highly secure form on the hard drive, and the passphrase is not stored.

Reminder:
The directory for storing this file is defined by passing the **KEY_DIRECTORY** parameter to the **setParameter()** method.

5.8.2 Encrypting Buffers With RSA

Method	**encryptBuffer**		
Description	Encrypts a buffer with RSA, using the public key whose Key ID is passed as an argument.		
Arguments	String	sKey_ID	Key_ID for the public key.
	byte [ ]	bBuffer	Buffer to be encrypted with RSA.
Return value	byte [ ]		Encrypted buffer.

Notes:

The buffer encryption/decryption functions have the following characteristics in SafeAPI Version 1.07:

- The maximum buffer size is 128 bits.
- The mode used is ECB (Electronic Code Book).
- The padding used is PKCS#7.

Method	**decryptBuffer**		
Description	Decrypts a buffer encrypted using the RSA public key whose Key ID is passed as an argument.		
Arguments	String	sKey_ID	Key_ID for the private key.
	char [ ]	caPassphrase	Passphrase corresponding to the private key with the same Key ID used to encrypt the buffer.
	byte [ ]	bBuffer	Buffer *encrypted with* ***encryptBuffer*** and now to be decrypted.
Return value	byte [ ]		Decrypted buffer.

5.8.3 Combination Asymmetric/Symmetric File Encryption With RSA: Principles

Encrypting files with the RSA algorithm combines both asymmetric and symmetric encryption. This method provides the benefits of asymmetric encryption (public/private keys) while retaining the speed offered by symmetric algorithms.

When encrypting a file for a user for whom there is a public RSA key:

- The pseudo-random number generator creates a symmetric key called the "session key". This will only be used to encrypt this file and will never be used again.
- The file is encrypted using a symmetric algorithm and the session key.
- The session key is itself encrypted with the asymmetric algorithm (RSA), using the public key for the user.
- The encrypted file + encrypted session key are merged into a new file.

The reverse procedure is followed to decrypt the file:

- The merged encryption file + session key is received.
- The encrypted session key is extracted and decrypted using the user's private RSA key.
- The session key is then used to decrypt the file.

5.8.4 Managing Recipient Lists

SafeAPI can handle file encryption for multiple users in one operation.

This is how it is done:

- Create a List of recipients (called the "List" below).
- Add RSA public key identifiers to the List. This corresponds to "adding Recipients to the List".
- Encrypt the file in one operation, by passing the List name to an API function as an argument. The file is then encrypted for all the Recipients.

Method	**createRecipients**		
Description	Creates a List of recipients for an RSA file encryption.		
Arguments	String	sListName	Name of the List of recipients.

Method	**addRecipient**		
Description	Adds a recipient to a List		
Arguments	String	sListName	Name of the List of recipients.
	String	sKey_ID	Key_ID for the recipient's public key.
Return value	Boolean	True: the recipient was added successfully False: if not	

5.8.5 Asymmetric File Encryption

Method	**decryptFile**		
Description	Encrypts a file using RSA, for the list of recipients passed as an argument.		
Arguments	String	sListName	Name of the List of recipients.
	String	sInputPath	Complete name of the file to be encrypted.
	String	sOutputPath	Complete name of the encrypted file to be created.

Notes:

- **sInputPath** and **sOutputPath** must be different.

Method	**decryptFile**		
Description	Decrypts a file encrypted using a public key and the method **decryptFile**		
Arguments	String	sKey_ID	Key_ID for the decryption private key.
	char [ ]	caPassphrase	Passphrase associated with the private key.
	String	sInputPath,	Complete name of the encrypted file.
	String	sOutputPath	Complete name of the decrypted file to be created.

5.8.6 Signature & Signature Verification

Notes:

- **1)** The signature and signature verification methods are used in pairs:

Signature method	**Corresponding Verification Method**
signBuffer()	verifyBuffer()
signFile() Version with complete file name	verifyFile() Version with complete file name
rawSignFile() Version with binary signature	rawVerifyFile() Version with binary signature
encryptAndSign() Asymmetric and signed encryption of the file.	decryptAndSign() Asymmetric decryption of a file and verification of its signature.

- **2) signBuffer()** and **verifyBuffer()** are used to sign/verify buffers.
- **3) signFile()** and **verifyFile()** rely on a signature contained in a file.

- **4) rawSignFile()** and **rawVerifyFile()** work with a binary signature whose management is left up to the programmer.

- **5)** The **encryptAndSign() / decryptAndSign()** methods allow encryption + signature in one operation. In this case, the signature is always stored in the encrypted file.
  The **isFileSigned** method is used to find out whether a file encrypted with SafeAPI has also been signed with a private key.

Method	**signBuffer**		
Description	Calculates and returns the signature associated with the specified buffer, using the private key passed as a parameter.		
Arguments	String	sKey_ID	Identifier for the private key to be used to sign.
	char [ ]	caPassphrase	Passphrase for the private key
	byte [ ]	Bbuffer	Buffer to be signed
Return value	byte [ ]	Value of the signature (series of bits) for bBuffer.	

Method	**verifyBuffer**		
Description	Verifies whether a binary signature associated with a buffer is valid.		
Arguments	String	sKey_ID	Identifier for the public key corresponding to the signature.
	String	BBuffer	Complete name of the buffer to be verified.
	byte [ ]	bSignature	Signature for bBuffer.
Return value	Boolean	True: the signature was verified (bBuffer authenticated) False: the signature was invalid.	

Method	**rawSignFile**		
Description	Calculates and returns the signature associated with the specified file, using the private key passed as a parameter.		
Arguments	String	sKey_ID	Identifier for the private key to be used to sign.
	char [ ]	caPassphrase	Passphrase for the private key
	String	sFilePath	Complete name of the file to be signed.
Return value	byte [ ]	Value of the signature (series of bits)	

Method	**rawVerifyFile**		
Description	Verifies whether a binary signature associated with a file is valid.		
Arguments	String	sKey_ID	Identifier for the public key corresponding to the signature.
	String	sFilePath	Complete name of the file to be verified.
	byte [ ]	bSignature	Signature for the FilePath file
Return value	Boolean	True: the signature was verified (file authenticated) False: the signature was invalid.	

Method	**signFile**		
Description	Calculates and saves in a file the signature associated with the specified file, using the private key passed as a parameter.		
Arguments	String	sKey_ID	Identifier for the private key to be used to sign.
	String	caPassphrase	Passphrase for the private key.
	String	sFilePath	Complete pathname for the file to be signed.
	String	sSigFilePath	Complete pathname for the file signature.

Method	**verifyFile**		
Description	Verifies whether the signature associated with the file is valid.		
	String	sKey_ID	Identifier for the public key corresponding to the signature.
Arguments	String	FilePath	Complete pathname for the file to be verified.
	String	sSigFilePath	Complete pathname for the file containing the signature for the FilePath file.
Return value	Boolean	True: the signature was verified (file authenticated) False: the signature was invalid	

Method	**encryptAndSign**		
Description	Signs the specified file using the private key, and encrypts it for the list of recipients.		
Arguments	String	sKey_ID	Identifier for the public key corresponding to the signature.
	char [ ]	caPassphrase	Passphrase for the private key used for the signature
	String	sListName	Name of the List of recipients
	String	sInputPath	Complete pathname for the file to be processed
	String	sOutputPath	Complete pathname for the output file

Method	**decryptAndVerify**		
Description	Decrypts the specified file using the private key, and verifies the signature.		
Arguments	String	sKey_ID	Identifier for the private decryption key.
	char [ ]	caPassphrase	Passphrase for the private decryption key.
	String	sSignKeyId	Identifier for the public key corresponding to the signature
	String	sInputPath	Complete pathname for the encrypted file to be processed
	String	sOutputPath	Complete pathname for the (unencrypted) output file
Return value	Int	1: the file was signed and was authenticated -1: the file was signed but not authenticated 0: the file was not signed	

Method	**isFileSigned**		
Description	Verifies that an encrypted file contains a signature.		
Arguments	String	FilePath	Complete pathname for the encrypted file.
Return value	Boolean	True: the encrypted file is signed False: the encrypted file is not signed.	

Method	**getSignKeyIdDigest**		
Description	Returns the hash value of the identifier for the key used to sign the file.		
Arguments	String	sFilePath	Complete pathname for the file
Return value	byte [ ]	The hash value of the identifier for the sender's key	

Note:

- For future use.

5.9 Class CryptoAsymRawRSA: methods of asymmetric encryption using “native” RSA keys

CryptoAsymRawRSA is intended for experienced users who want to use advanced RSA functions.

Use of the CryptoAsym class is recommended in most cases.

The methods described in this chapter all use RSA keys in a “native” format, i.e. the keys are expressed as numbers.

Note:

The values of the private keys are not protected in any of the methods in chapter 5.9.

They must remain secret to ensure the security of the encryption.

5.9.1 Encrypting buffers with “native” RSA keys

Method	encryptBufferRawKey		
Description	Encrypts a buffer with RSA, passing the “native” form of a public key as an argument.		
Arguments	byte[ ]	bN_Modulus	Public modulus N of the RSA key. Number expressed in bits in twos complement in *big-Endian.*
	byte[ ]	bE_Exponent	Public exponent E. Number expressed in bits in twos complement in *big-Endian.*
	byte[ ]	bBuffer	Buffer to be encrypted with RSA.
Return value	byte[ ]		Encrypted buffer.

Mode & Padding

The buffer encryption/decryption functions have the following characteristics:

- The **mode** used is **ECB** (Electronic Code Book).
- The **padding** used is **PKCS#7.**

5.9.2 Decrypting buffers using “native” RSA keys

Method	**decryptBufferRawKey**		
Description	Decrypts an RSA encrypted buffer, passing the “native” form of the **private** key as an argument.		
Arguments	byte[ ]	bD_Exponent	**D**, **Private** exponent to be kept secret. Number expressed in bits in twos complement in *big-Endian.*
	byte[ ]	bP_Factor	1st factor **Private P** to be kept secret. (prime number). Number expressed in bits in twos complement in *big-Endian.*
	byte[ ]	bP_Factor	2nd factor **Private Q** to be kept secret. (prime number). Number expressed in bits in twos complement in *big-Endian.*
	byte[ ]	bBuffer	RSA encrypted buffer to be decrypted.
Return value	byte[ ]		Decrypted buffer.

Note/Reminder:

The 3 values bD_Exponent, bP_Factor and bP_Factor are the “clear” components of the private key and must remain confidential.

Mode & Padding

The buffer encryption/decryption functions have the following characteristics:

- The **mode** used is **ECB** (Electronic Code Book).
- The **padding** used is **PKCS#7.**

5.9.2.1 Initializing and loading SafeAPI format RSA keys into memory

The two functions **loadPublicKey** and/or **loadPrivateKey** enable the extractions to be prepared **and must be launched prior to** the other functions in order to access the components.

Method	**loadPublicKey**		
Description	Loads into memory the public key corresponding to a key generated with SafeAPI.		
Arguments	String	sKey_ID	Key_ID of the public key.
Return value			

Method	**loadPrivateKey**		
Description	Loads into memory the private key corresponding to a key generated with SafeAPI.		
Arguments	String	sKey_ID	Key_ID of the private key.
	char[ ]	caPassphrase	Passphrase protecting the key.
Return value			

5.9.2.2 Methods for getting components of RSA keys

Before launching the **loadPublicKey and/or loadPrivateKey** functions, the following “get” type functions allow the various components of the RSA key pair to be retrieved as a byte table or a hexadecimal string:

Method	**getPublicKeyExponent**	
Description	Gets the public exponent E of the public key	
Arguments	None	
Return value	byte[ ]	Public exponent E. Number expressed in bits in twos complements in *big-Endian.*

Method	**getHexPublicKeyExponent**	
Description	Gets the public exponent E of the public key as a hexadecimal string.	
Arguments	None	
Return value	String	Public exponent E as a hexadecimal string.

Method	**getModulus**	
Description	Gets the public modulus N of the RSA key	
Arguments	None	
Return value	byte[ ]	Public modulus N of the RSA key. Number expressed in bits in twos complement in *big-Endian.*

Method	**getHexModulus**	
Description	Gets the public modulus N of the RSA key as a hexadecimal string.	
Arguments	None	
Return value	String	Public modulus N of the RSA key as a hexadecimal string.

Method	**getPrivateKeyExponent**	
Description	Gets the **Private exponent D** to be kept secret.	
Arguments	None	
Return value	byte[ ]	**D**, **Private** exponent to be kept secret. Number expressed in bits in twos complements in *big-Endian.*

Method	**getHexPrivateKeyExponent**	
Description	Gets the **Private exponent D** to be kept secret as a hexadecimal string.	
Arguments	None	
Return value	String	**D**, **Private** exponent to be kept secret as a hexadecimal string.

Method	**getP**	
Description	Gets the 1st factor **Private P** to be kept secret (prime number). Number expressed in bits in twos complement in *big-Endian.*	
Arguments	None	
Return value	byte[ ]	1st factor **Private P** to be kept secret. (prime number). Number expressed in bits in twos complement in *big-Endian.*

Method	**getHexP**	
Description	Gets the 1st factor **<u>Private</u> P** to be kept secret (prime number) as a hexadecimal string.	
Arguments	None	
Return value	String	1st factor **<u>Private</u> P** to be kept secret (prime number) as a hexadecimal string.

Method	**getQ**	
Description	Gets the 2nd factor **<u>Private</u> Q** to be kept secret (prime number). Number expressed in bits in twos complement in *big-Endian.*	
Arguments	None	
Return value	byte[ ]	1st factor **<u>Private</u> P** to be kept secret (prime number) as a hexadecimal string.

Method	**getHexQ**	
Description	Gets the 2nd factor **<u>Private</u> Q** to be kept secret (prime number) as a hexadecimal string.	
Arguments	None	
Return value	String	2nd factor **<u>Private</u> Q** to be kept secret (prime number) as a hexadecimal string.

Method	**getInverseOfQModP**	
Description	Gets **U**, the mathematical inverse of Q modulo P to be kept secret. Number expressed in bits in twos complement in *big-Endian.*	
Arguments	None	
Return value	byte[ ]	**U**, inverse of Q modulo P to be kept secret. Number expressed in bits in twos complement in *big-Endian.*

Method	**getHexInverseOfQModP**	
Description	Gets **U**, inverse of Q modulo P to be kept secret as a hexadecimal string.	
Arguments	None	
Return	String	**U**, inverse of Q modulo P to be kept secret as a hexadecimal string.

5.9.2.3 Full example of a Java implementation

A full example of exporting RSA keys as components and encrypting and decrypting the buffer can be found in the program: **RawRSAExemple.java** in the samples directory.

5.9.3 Direct application of RSA encryption

CryptoAsymRawRSA allows RSA to be directly used on a buffer, by using the various components of the key.

This direct use allows for “non-traditional” operations, not accepted by the other SafeAPI methods:

- Encryption of a buffer with a private key.
- Decryption of a buffer with a public key.

Method	**rsaWithCrt**		
Description	Directly applies the RSA algorithm to a buffer by using the components of the **private** key. **rsaWithCrt** uses the Chinese Remainder Theorem (CRT)		
Arguments	byte[ ]	bD_Exponent	**D**, **<u>Private</u>** exponent to be kept secret. Number expressed in bits in twos complement in *big-Endian.*
	Byte[ ]	bP_Factor	1st factor **<u>Private</u> P** to be kept secret. (prime number). Number expressed in bits in twos complement in *big-Endian.*
	byte[ ]	bP_Factor	2nd factor **<u>Private</u> Q** to be kept secret. (prime number). Number expressed in bits in twos complement in *big-Endian.*
	byte[ ]	bU	**U**, inverse of Q modulo P to be kept **secret**. Number expressed in bits in twos complement in *big-Endian.*
	byte[ ]	bBuffer	Buffer to be encrypted using RSA.
Return value	byte[ ]		Buffer encrypted using RSA.

Note/Reminder:

<u>The 4 values bD_Exponent, bP_Factor, bP_Factor and bU represent “clear” components of the private key and must remain confidential.</u>

Method	**rsa**		
Description	Directly applies the RSA algorithm to a buffer by using the components of the public key. **rsa** does not use the Chinese Remainder Theorem (CRT)		
Arguments	byte[ ]	bN_Modulus	Public modulus N of the RSA key. Number expressed in bits in twos complements in *big-Endian.*
	byte[ ]	bE_Exponent	Public exponent E. Number expressed in bits in twos complement in *big-Endian.*
	byte[ ]	bBuffer	Buffer to be encrypted using RSA.
Return value	byte[ ]		Buffer encrypted using RSA.

Note for the two rsa methods:

- No mode is applied.
- The **rsa** methods break the buffer down into blocks of bytes of the same size as the key divided by 8. For example, if it is a 2048 bit key, the resulting buffer is made up of 2048 / 8 blocks = 256 bytes.
- Any encryption below the size of a block generates an encrypted block of the same size as a block.
- Example: using a 2048 bit key, we want to apply RSA to a buffer of 1038 bytes. 1038 = 256 + 256 + 256 + 256 + 14. Each encrypted block will be 256 bytes long. The resulting buffer will thus be 1280 bytes = 256 + 256 + 256 + 256 + 256, the final 14 byte piece being topped up to 256 bytes after encryption.

5.9.3.1 Full example of a Java implementation

A full example of exporting RSA keys as components and encrypting and decrypting the buffer can be found in the program: **TestRSA.java** in the samples directory.

5.10 Class CryptoDir: methods for encrypting directories

The methods of the **CryptoDir** class can be broken down into three main groups.

- 1) Methods related to the dynamic parameterization of the API.
- 3) Methods of symmetrically encrypting directories
- 3) Methods of asymmetrically encrypting directories.

5.10.1 Dynamic parameterization of CryptoDir

The dynamic parameterization of CryptoDir is done using the **setParameter() and getParameter()** functions, described in **5.3.1 CryptoCommon :**

There are additional parameters:

CryptoDir parameter	**Role & range of values**

`DIR_DELETE_LEVEL`	Type of file delete after encrypting the directory.
	`DELETE_NONE   : no delete.`
	`DELETE_SIMPLE : normal delete.`
	`DELETE_WIPE_1 : level 1 wipe (weak)`
	`DELETE_WIPE_2 : level 2 wipe (medium)`
	`DELETE_WIPE_3 : level 3 wipe (strong)`

`VERBOSE`	Indicates if file encryption must be displayed on the system screen. True or False
`NB_RETRY_WHEN_LOCK`	Number of successive attempts to encrypt a file locked by another process.
`SECONDS_PAUSE_WHEN_LOCK`	Number of seconds of pause between each attempt at encrypting a file locked by another process.

5.10.2 List of the default parameter values

CryptoDir **parameter**	**Default values**
`DIR_DELETE_LEVEL`	`DELETE_SIMPLE`
`VERBOSE`	True
`NB_RETRY_WHEN_LOCK`	3
`SECONDS_PAUSE_WHEN_LOCK`	1

5.10.3 Symmetric encryption of directories

Method	**encryptDirWithPassphrase**		
Description	Encrypts all the files in a directory. The encrypted files are placed in a different output directory, to be specified. The symmetric encryption key is generated dynamically by deriving it from the passphrase.		
Arguments	String	sAlgorithm	Algorithm to be used. (Cf. the list of allowed values in SAPIDOC].).
	char[ ]	caPassphrase	Passphrase protecting the directory.
	String	sInputDir	Full name of the directory to encrypt.
	String	sOutputDir	Full name of the output directory for the encrypted files.

Notes:

- **sInputDir** and **sOutputDir** must be different.

- Reminder: The term **full name** means the full name of a directory with the units.
  - Windows example: **`c:\safelogic\lib`**
  - Unix example : **`/home/safelogic/lib`**

- The files can be deleted from **sInputDir** depending on the value of the **`DIR_DELETE_LEVEL`** parameter.

- The two parameters NB_RETRY_WHEN_LOCK and SECONDS_PAUSE_WHEN_LOCK do not apply to this API.

Method	**decryptDirWithPassphrase**		
Description	Decrypts all the files in a directory. The encrypted files must have been encrypted using **encryptDirWithPassphrase**. The decrypted files are placed in a different output directory, to be specified.		
Arguments	String	sAlgorithm	Algorithm to be used. (Cf. the list of allowed values).
	char[ ]	caPassphrase	Passphrase protecting the directory (the one that was used with **encryptDirWithPassphrase**).
	String	sInputDir	Full name of the directory to decrypt.
	String	sOutputDir	Full name of the output directory for the decrypted files.

Notes:

- Cf. the notes to **encryptDirWithPassphrase().**

Method	**encryptDir**		
Description	Encrypts all the files in a directory. The encrypted files are placed in a different output directory, to be specified. encryptDir uses a symmetric key generated with **CryptoSym.genSecretKey()**		
Arguments	String	sKey_ID	Key_ID of the key.
	char[ ]	caPassphrase	Passphrase protecting the sKey_ID key.
	String	sInputDir	Full name of the directory to be encrypted.
	String	sOutputDir	Full name of the output directory for the encrypted files.

Notes:

- **sInputDir** and **sOutputDir** must be different.
- The files can be deleted from **sInputDir** depending on the value of the **`DIR_DELETE_LEVEL`** parameter.
- The two parameters NB_RETRY_WHEN_LOCK and SECONDS_PAUSE_WHEN_LOCK do not apply to this API.

Method	**decryptDir**		
Description	Decrypts all the files in a directory. The encrypted files must have been encrypted with **encryptDir** The decrypted files are placed in a different output directory, to be specified.		
Arguments	String	sKey_ID	Key_ID of the key.
	char[ ]	caPassphrase	Passphrase protecting the sKey_ID key.
	String	sInputDir	Full name of the directory to be decrypted.
	String	sOutputDir	Full name of the output directory for the encrypted files.

Notes:

- Cf. the notes to **encryptDir().**

5.10.4 Asymmetric directory encryption

Asymmetric directory encryption uses the principles of **combination encryption** and of the **management of recipient lists** described in **5.8.4 Managing Recipient Lists**

The encryption principles are the same as those applied for the asymmetric file encryption:

- **CryptoAsym.encryptFile().**
- **CryptoAsym.decryptFile().**

In fact, for every file to be encrypted, **asymEncryptDir()** calls **CryptoAsym.encryptFile()** and, for every file to be decrypted **asymDecryptDir()** calls **CryptoAsym.decryptFile(),**

The functions **asymEncryptDir()** and **asymDecryptDir()** use RSA keys generated by **CryptoAsym.genKeyPair().**

Method	**asymEncryptDir**		
Description	Uses RSA to Encrypt all the files in a directory for a recipient list passed as an argument. The encrypted files are placed in a different output folder, to be specified.		
Arguments	String	sListName	Name of the recipient list.
	String	sInputDir	Full name of the directory to encrypt.
	String	sOutputDir	Full name of the output folder for the encrypted files.

Notes:

- **sInputPath** and **sOutputPath** must be different.
- The files can be deleted from **sInputDir** depending on the value of the **`DIR_DELETE_LEVEL`** parameter.
- The two parameters NB_RETRY_WHEN_LOCK and SECONDS_PAUSE_WHEN_LOCK make it possible to define new attempts for each file locked by another process.

Method	**asymDecryptDir**		
Description	Decrypts all the files in a directory. The encrypted files must have been encrypted using **encryptDir** The decrypted files are placed in a different output directory, to be specified.		
Arguments	String	sKey_ID	Key_ID of the private encryption key.
	char[ ]	caPassphrase	Passphrase associated with the private key.
	String	sInputDir	Full name of the directory to be decrypted.
	String	sOutputDir	Full name of the output directory for the decrypted files.

Notes:

- Cf. the notes to **asymEncryptDir().**

6 Using The Methods

6.1 Using The Hashing and Error Handling Methods

There are three steps to follow when hashing a buffer:

- 1) First initialize the hash tool using **hashDataBufferInit**.
- 2) Then fill the buffer by successive calls to **hashDataBuffer**.
- 3) When all the information (bytes) has been sent to the buffer, calculate and retrieve the hash value using **hashDataBufferDigest**.

Now that we have presented all the cryptography methods of the API, we will describe the general mechanism to use when calling a cryptography function of the API in order to include error handling.

For any action you perform, you must verify that the operation was successful. This is done by calling **isOperationOK** and, if needed, diagnosing the problem using **getRegisteredError**.

It is also possible to retrieve a complete account of the error using **getRawError** (this is only recommended if the diagnosis returned by **getRegisteredError** indicates an unknown error).

6.2 Using The Symmetric Encryption Methods

Encryption must be preceded by generating a key with **genSecretKey.**

The resulting secret key is stored in a file and associated with a "passphrase" that protects it. The passphrase itself is converted into a key that is used to encrypt the secret key.

The generated key is valid for an indefinite period. It is therefore possible to separate the VB programs for key generation, encryption, and decryption by using the methods:

- **encryptBuffer** or **encryptFile** for encryption.
- **decryptBuffer** or **decryptFile** for decryption.

6.3 Using The Asymmetric Encryption Methods

First you must generate a binary file containing a series of random data. These data are extremely important, as they are used as a *pseudo-random* database for the generation of symmetric 128-bit session keys.

The API to use is **createSeedFile**.

The second step consists of generating a pair of RSA keys using **genKeyPair.**

These two RSA keys (private, public) are stored in two separate files. The private key is symmetrically encrypted, and is associated with a passphrase that protects it and must always remain confidential.

The next step consists of generating a list of recipients by using:

- **createRecipients**
- **addRecipient**

Then the encryption/decryption functions can be used:

- **decryptFile**
- **decryptFile**

as well as the signature and signature verification functions:

- **signFile / rawSignFile**
- **verifyFile / rawVerifyFile**

It is also possible to do asymmetric encryption + signature operations in one step:

- **encryptAndSign**
- **decryptAndSign**

See the examples described in detail later in this guide:

- Java program: **TestPKI.java**
- VB project: **SafeAPI_VB**

6.4 Sample Java Programs

SafeAPI comes with sample Java programs. Some of these are run in command line mode.

Name of sample Java program	Description
`TestHash.java`	Hashes a buffer.
`TestHashFile.java`	Hashes a file.
`TestSym.java`	• Generates symmetric keys with Blowfish, CAST-128, and IDEA. • Symmetric encryption of a buffer. • Symmetric encryption of a file.
`TestRSACrypt.java`	• Generates a pair of RSA keys. • Asymmetric encryption/decryption of a buffer. • Asymmetric encryption/decryption of a file.
`TestRSASign.java`	• Signs a buffer. • Signs a file (signature included or detached). • Asymmetric encryption and signing of a file.

Notes:

- To run these programs, prefix their name with the Java package **`com.safeapi.test`**

- Windows example:

```
c:\> java com.safeapi.test.TestHash
```

- Unix example:

```
$ java com.safeapi.test.TestHash
```

6.5 Visual Basic Examples: Complete SafeAPI_VB Project

SafeAPI comes with the complete Visual Basic 6.0 project **SafeAPI_VB**, which contains a series of windows and procedures for generating a file of seed data, generating symmetric or RSA keys, hashing a buffer or file, encrypting a file, etc.

Note: the programs, projects, VB windows, etc. are provided as examples only and are not supported by SafeLogic.

7 IMPLEMENTATION/CRYPTOGRAPHY FAQ

7.1 Purpose of this FAQ

This FAQ explains how to validate or "prove" the quality of the implementation of the cryptography algorithms in SafeAPI. It assumes that you are familiar with basic cryptography concepts.

7.2 How can I test the implementation of MD5?

RFC 1321 defines a set of tests for MD5.
See http://www.faqs.org/rfcs/rfc1321.html

Here is the complete set. The result corresponds to a conversion of the binary result into a hexadecimal string:

```
MD5 ("")    = d41d8cd98f00b204e9800998ecf8427e
MD5 ("a")   = 0cc175b9c0f1b6a831c399e269772661
MD5 ("abc") = 900150983cd24fb0d6963f7d28e17f72
MD5 ("message digest") = f96b697d7cb7938d525a2f31aaf161d0
MD5 ("abcdefghijklmnopqrstuvwxyz") = c3fcd3d76192e4007dfb496cca67e13b
MD5 ("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789")
      = d174ab98d277d9f5a5611c2c9f419d9f
MD5 ("123456789012345678901234567890123456789012345678901234567890123456
78901234567890") = 57edf4a22be3c955ac49da2e2107b67a
```

7.3 How can I test the implementation of SHA-1?

FIPS 180-1 defines two tests for SHA-1.
See http://www.itl.nist.gov/fipspubs/fip180-1.htm

```
SHA-1( "abc") =  "A9993E364706816ABA3E25717850C26C9CD0D89D
SHA-1( "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq")
           = "84983E441C3BD26EBAAE4AA1F95129E5E54670F1"
```

7.4 How can I test the implementation of Blowfish?

A test set recommended by the author of Blowfish is defined at ftp://ftp.psy.uq.oz.au/pub/Crypto/Blowfish/test.data

Blowfish is also described in detail at:
http://www.counterpane.com/blowfish.html

In SafeAPI, Blowfish is implemented in CFB (Cipher Feedback) mode without padding.

You can run the CFB test:

Object/data	**Hexadecimal value**
Key in hexadecimal:	`0123456789ABCDEFF0E1D2C3B4A59687`
IV (Initialization Vector) in hex:	`FEDCBA9876543210`
Unencrypted hex string:	`37363534333231204E6F77206973207468652074696D6520666F722000`
Encrypted hex string:	`E73214A2822139CAF26ECF6D2EB9E76E3DA3DE04D1517200519D57A6C3`

7.5 How can I test the implementation of CAST-128?

RFC 2144 defines a test set for CAST, but only in ECB mode.
See http://www.faqs.org/rfcs/rfc2144.html.

CAST-128 is implemented in CFB mode in SafeAPI, so there is no "standard" test set.

7.6 How can I test the implementation of IDEA?

The IDEA tests are defined in a file that can be downloaded from ASCOM at:
http://www.ascom.ch/infosec/downloads.html

IDEA is implemented in CFB in SafeAPI. Here are two tests taken from the test set for CFB mode:

Object/data	Hexadecimal value
Hexadecimal key:	`729A27ED8F5C3E8BAF16560D14C90B43`
IV (Initialization Vector) in hex:	`C121A1B050D8286C`
Test 1	
String C1 unencrypted hex:	`D53FABBF94FF8B5F`
String C1 encrypted hex:	`E42323CAF932B933`
Test 2	
String C2 unencrypted hex:	`94FF8B5F`
String C2 encrypted hex:	`A5E3032A`

7.7 How are pseudo-random numbers generated with SafeAPI?

The pseudo-random numbers are generated using the ANSI X9.17 algorithm.

X9.17 is a powerful random number generator that takes a pair of values (random value, key) and encrypts it using a key provided by the user and a symmetric algorithm. It produces a new pair (random value, key) that cannot be predicted by an attacker.

7.8 How is the IV generated in CFB with SafeAPI?

The IV (Initialization Vector) for CFB mode is generated using the ANSI X9.17 algorithm.

7.9 How are the secret keys stored?

Before the secret key is saved on the hard drive, it is encrypted using:

- The algorithm selected by the user in the API: Blowfish, CAST-128, IDEA.
- CFB mode.
- A 128-bit key that is the MD5 hash value of the passphrase parameter passed to **genSecretKey.**

The secret keys are never stored unencrypted on the hard drive.

7.10 How are the RSA private keys stored?

Before the private key is saved on the hard drive, it is encrypted using:

- The Blowfish algorithm.
- CFB mode.
- A 128-bit key that is the MD5 hash value of the passphrase parameter passed to **genKeyPair.**

The RSA private keys are never stored unencrypted on the hard drive.

8 SUPPORT

If you have any technical questions, contact us at:

SafeLogic
27/29, rue Raffet
75016 Paris
Tel : (33) (0)1 45 72 25 15
Fax: (33) (0)1 45 72 14 06
E mail: contact@safelogic.com
Web : http://www.safelogic.com
