libsecure Overview

Table of Contents

About This Document 3
Conventions 3
Errata 3
1 **Library Overview 4**
Overview 4
Related Files 4
Sample Programs 4
2 **Using the Library 5**
Overview 5
Initialization Process 5
Cryptography 6
Message Authentication Process 10
Library Termination 11

About This Document

The purpose of this document is to provide an overview of the libsecure library and describe its use from a developer's point of view.

Conventions

The typographical conventions used in this guide are explained in this section.

Hints

A GUI shortcut or other useful tip for gaining maximum use from the software is presented as a 'hint' surrounded by a box. For example:

> **Hint:** Example hint.

Notes

Additional advice or related information is presented as a 'note' surrounded by a box. For example:

> **Note:** Example note.

Text

File names, source code and command-line text are formatted in a fixed-width font. For example:

```
SceLibSecureErrorType error;
```

Errata

Any updates or amendments to this guide can be found in the release notes that accompany the release.

1 Library Overview

This chapter describes the libsecure library.

Overview

The libsecure library is a cross-platform API that may be used on PSP™ (PlayStation®Portable), PlayStation®3, PlayStation®Vita, and PlayStation®4. It provides cryptography functionalities (encryption and decryption methods) and security functionalities such as hash calculation and message authentication.

The ciphers supported by the library are AES, Blowfish, TEA, XTEA, DES, Triple DES and RSA. The hash methods supported by the library are MD5, SHA-1, SHA-256, SHA-384, and SHA-512.

Related Files

To use libsecure library, you must have the files listed in Table 1.

Table 1 Prerequisite Files

File Name	Description	Directory
`libsecure.h`	Header file	`include`
`libSceSecure.a`	Static library file	`Lib`
`libSceSecure_stub.a`	Stub library file	`Lib`
`libSceSecure.suprx`	Library module file	`sce_module`

You can use the libsecure by linking the static library file within your build, or by linking against the stub library file and loading the module into your application before using it.

Sample Programs

Four samples are provided to illustrate the usage of the libsecure library. They are available in the `Samples` directory of the libsecure package, as follows:

sample_code/system/api_libsecure/cipher/

This sample program demonstrates the cryptography functionality API usage. It performs the encryption and decryption of a single message block using the libsecure API functions.

sample_code/system/api_libsecure/cipherfw/

This sample program demonstrates how to expand the cryptography methods. It creates additional ciphers that can be used for the encryption and decryption of a single message block.

sample_code/system/api_libsecure/hash/

This sample program demonstrates the security functionality API usage. It performs the hash calculation and the authentication of a message using the libsecure API functions.

sample_code/system/api_libsecure/hashfw/

This sample program demonstrates how to expand the hash methods. It creates additional hash methods that can be used for the hash calculation and authentication of a message.

2 Using the Library

This chapter uses a walkthrough to describe how to use the libsecure library from a developer's point of view.

Overview

The following sections describe a walkthrough that illustrates the most basic usage of the library. The purpose of the walkthrough is to encrypt and decrypt a message and to ensure that a message has not been tampered with. The walkthrough has four parts:

- Initialization Process
- Cryptography
- Message Authentication Process
- Library Termination

Initialization Process

The first step in using the library is to call the `sceLibSecureInit` function. This requires a memory block provided with the structure `SceLibSecureBlock` and some flags detailing which functionalities of the library are used.

The memory block should be at least 72 bytes (when using only one cipher method and one hash method). However, to use the libsecure library to its full potential the recommended size is 512 bytes.

The flags define what functionality can be used after the initialization. The flags are described in Table 2.

Table 2 Initialization Flags

Flag	Description
`SCE_LIBSECURE_FLAGS_CIPHER_TEA`	Specify this flag if you need the TEA cipher for encrypting messages.
`SCE_LIBSECURE_FLAGS_CIPHER_XTEA`	Specify this flag if you need the XTEA cipher for encrypting messages.
`SCE_LIBSECURE_FLAGS_CIPHER_BLOWFISH`	Specify this flag if you need the Blowfish cipher for encrypting messages.
`SCE_LIBSECURE_FLAGS_CIPHER_AES`	Specify this flag if you need the AES cipher for encrypting messages.
`SCE_LIBSECURE_FLAGS_CIPHER_RSA`	Specify this flag if you need the RSA cipher for encrypting messages.
`SCE_LIBSECURE_FLAGS_CIPHER_DES`	Specify this flag if you need the DES cipher for encrypting messages.
`SCE_LIBSECURE_FLAGS_CIPHER_TDEA`	Specify this flag if you need the TDEA cipher for encrypting messages.
`SCE_LIBSECURE_FLAGS_HASH_SHA1`	Specify this flag if you need the SHA1 hash methods for message authentication.
`SCE_LIBSECURE_FLAGS_HASH_MD5`	Specify this flag if you need the MD5 hash methods for message authentication.
`SCE_LIBSECURE_FLAGS_HASH_SHA256`	Specify this flag if you need the SHA256 hash methods for message authentication.

Flag	Description
SCE_LIBSECURE_FLAGS_HASH_SHA384	Specify this flag if you need the SHA384 hash methods for message authentication.
SCE_LIBSECURE_FLAGS_HASH_SHA512	Specify this flag if you need the SHA512 hash methods for message authentication.
SCE_LIBSECURE_FLAGS_RANDOM_GENERATOR	Specify this flag if you need to generate random messages.
SCE_LIBSECURE_FLAGS_RELAXED	Specify this flag if you need standard enforcements to be relaxed.
SCE_LIBSECURE_FLAGS_EVP_PADDING_COMPATIBILITY	Specify this flag if you require the same behavior for message padding as EVP when used in OpenSSL.

In this example, the library is initialized and flags are specified for random generator messages, AES encryption, and the SHA1 hash method:

```
SceLibSecureErrorType error;
SceLibSecureBlock runtimemem;

/* Allocate a 1KB memory block */
runtimemem.length = 1*1024;
runtimemem.pointer = malloc(runtimemem.length);
if(runtimemem.pointer != NULL)
{
    /* Initialize the library */
    error = sceLibSecureInit(SCE_LIBSECURE_FLAGS_CIPHER_AES |
        SCE_LIBSECURE_FLAGS_HASH_SHA1 |
        SCE_LIBSECURE_FLAGS_RANDOM_GENERATOR, &runtimemem);
    ...
}
```

Cryptography

Ciphers

The libsecure library supports symmetric and asymmetric ciphers. The ciphers included in the library are listed in Table 3.

Table 3 Ciphers

Cipher	Name	Type
AES	Advanced Encryption Standard	Symmetric
Blowfish	Blowfish	Symmetric
TEA	Tiny Encryption Algorithm	Symmetric
XTEA	eXtended TEA	Symmetric
DES	Data Encryption Standard	Symmetric
TDEA	Triple DES	Symmetric
RSA	Ron Rivest, Adi Shamir and Leonard Adleman	Asymmetric

The library manages different block cipher modes and does not depend on the cipher used for the encryption and decryption. You use a particular block cipher mode depending on the type of data access you require (sequential or direct access). The block cipher modes supported by the library are listed in Table 4.

Table 4 Block Cipher Modes

Mode	Name	Data Access
ECB	Electronic code book	Direct for encryption and decryption
CBC	Cipher block chaining	Sequential for encryption, direct for decryption
PCBC	Propagating cipher block chaining	Sequential for encryption and decryption
CFB	Cipher feedback	Sequential for encryption, direct for decryption
OFB	Output feedback	Sequential for encryption and decryption
CTR	Counter	Direct for encryption and decryption

Note: It is recommended that you do NOT use the ECB mode for a long sequence of data, or if you are encrypting and decrypting more than one block of data.

Padding

Ciphers work on units of a fixed size (which depends on the cipher used or the key size); however, messages vary in length. Therefore, the libsecure library manages padding to allow all messages to be encrypted and decrypted successfully. The mode of padding used depends mainly on the cipher or type of cipher used to encrypt and decrypt data. Multiple padding modes can be used together to pad a message. The padding modes supported by the library are listed in Table 5.

Table 5 Padding Modes

Padding	Description	Cipher Use
None	No padding is performed on the message and the remaining incomplete block is not encrypted or decrypted.	All symmetric ciphers.
Normal	No padding is performed on the message and the remaining incomplete block may be encrypted or decrypted (depends on the block cipher mode).	All symmetric ciphers. **Note:** the last incomplete block will be encrypted for CFB, OFB, and CTR block cipher modes.
Stealing	No padding is performed on the message and the remaining incomplete block is encrypted or decrypted using the stealing method.	All symmetric ciphers. **Note:** the last incomplete block will be encrypted for ECB, CBC, and PCBC block cipher modes.
Residual block termination	No padding is performed on the message and the remaining incomplete block is encrypted or decrypted using the CBC block cipher mode.	All symmetric ciphers.
Nils	The remaining incomplete block is filled with zero values.	All symmetric ciphers.
Spaces	The remaining incomplete block is filled with spaces.	All symmetric ciphers.
Random	The remaining incomplete block is filled with random values.	All symmetric ciphers.
Size	The remaining incomplete block is filled with values equal to the size of the incomplete block.	All symmetric ciphers.
Nil size	The remaining incomplete block is filled with zero values except the last one, which is the size of the incomplete block.	All symmetric ciphers.
Bit	The remaining incomplete block is filled with bit 0 except the last one, which is bit 1.	All symmetric ciphers.
PKCS#1	The padding used complies with the PKCS#1 v2.1 standard.	RSA cipher.

Padding	Description	Cipher Use
OAEP	The padding used complies with the Optimal Asymmetric Encryption Padding standard.	RSA cipher.

Setting the Context

A context may be needed depending on the cipher and the block cipher mode used before the encryption or decryption process.

To set the context:

(1) Retrieve the context size if one is required. Retrieving the context size requires the cipher used, the block cipher mode, the key, the number of rounds (only required for TEA, XTEA, and Blowfish ciphers):

```
u8 array_key[16]= { 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff, 0x00, 0x11,
0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99 };
SceLibSecureSymmetricKey key = { sizeof(array_key), array_key };
size_t context_size;
SceLibSecureErrorType error;

/* Retrieve the context size */
error = sceLibSecureCryptographyGetContextSize(SCE_LIBSECURE_CIPHER_AES,
     SCE_LIBSECURE_BLOCKCIPHERMODE_ECB, &key, 0, &context_size);
if(error != SCE_LIBSECURE_OK)
{
    /* Error management */
    ...
}
```

(2) Set the context according to the parameters passed when retrieving the context size. Depending on the block cipher mode used, you may also specify an initialization vector (IV) and a file offset (used for CTR only).

```
SceLibSecureErrorType error;
SceLibSecureBlock context;

/* Allocate the context */
context.length = context_size;
context.pointer = malloc(context.length);
if(context.pointer != NULL)
{
    /* Set the context */
    error = sceLibSecureCryptographySetContext(&context,
        SCE_LIBSECURE_CIPHER_AES, SCE_LIBSECURE_BLOCKCIPHERMODE_ECB,
        &key, 0, NULL, 0);
    ...
}
```

Encryption and Decryption

To perform encryption or decryption:

(1) Prepare the block or the message (succession of blocks) you want to encrypt or decrypt. The message or block length must match the cipher block size, which you can retrieve by calling the `sceLibSecureCryptographyGetBlockSize` function. This function requires the cipher, the key (`SceLibSecureSymmetricKey` for symmetric ciphers, and `SceLibSecureAsymmetricKey` for asymmetric ciphers) and the padding scheme used:

```
SceLibSecureErrorType error;
SceLibSecureSymmetricKey key = { sizeof(array_key), array_key };
size_t block_size;
```

```
/* Retrieve the block size */
error = sceLibSecureCryptographyGetBlockSize(SCE_LIBSECURE_CIPHER_AES,
     &key, SCE_LIBSECURE_PADDING_NONE, &block_size)
if(error != SCE_LIBSECURE_OK)
{
    /* Error management */
    ...
}
```

(2) Retrieve the block matching the block size or the message. The message must be a multiple of the block size.

> **Note:** If the message is not a multiple of the block size, you need to apply a padding scheme to the message before the encryption process. Refer to the *libsecure Reference* for information about this process.

(3) Finally, call the `sceLibSecureCryptographyEncrypt` or `sceLibSecureCryptographyDecrypt` function:

```
SceLibSecureErrorType error;
u8 array_message[message_size];
SceLibSecureMessage message = { array_message , sizeof(array_message) };

/* Generate a random message */
error = sceLibSecureRandom(&message);
if(error == SCE_LIBSECURE_OK)
{
    /* Encrypt the message */
    error = sceLibSecureCryptographyEncrypt(&context, &message,
       SCE_LIBSECURE_PADDING_NONE);
    if(error != SCE_LIBSECURE_OK)
    {
        /* Error management */
        ...
    }

    /* Decrypt the message */
    error = sceLibSecureCryptographyDecrypt(&context, &message,
       SCE_LIBSECURE_PADDING_NONE);
    if(error != SCE_LIBSECURE_OK)
    {
        /* Error management */
        ...
    }
}
```

Destroying the Context

It is good practice to destroy the previously used context, so that the data used to retrieve the original message is not used, or if you do not plan to reuse the same context to continue the encryption or the decryption process.

To destroy the context, use the `sceLibSecureCryptographyDeleteContext` function:

```
SceLibSecureErrorType error;

/* Destroy the context */
error = sceLibSecureCryptographyDeleteContext(&context);
...
```

Message Authentication Process

The encryption and decryption of a message may be affected if the message has been tampered with. Message authentication can be used to ensure a message is signed and has not been tampered with. To authenticate a message, the authentication process uses the hash functionality of the library, a key, and the message. The process returns an HMAC value, which is then used by the receiver of the message to ensure the message has not been tampered with.

The libsecure library supports a number of hash functions. Additional hash functions can be added to the library by the developer. The hash functions included in the library are listed in Table 6.

Table 6 Hash Functions

Hash	Name
MD5	MD5 Message Digest Algorithm
SHA-1	SHA-1 Secure Hash Algorithm
SHA-256	SHA-256 Secure Hash Algorithm
SHA-384	SHA-384 Secure Hash Algorithm
SHA-512	SHA-512 Secure Hash Algorithm

Note: A context is always required to use the hash functionality provided by the library.

Setting the Context

To set the context:

(1) Retrieve the context size and the digest size (which is the same as the HMAC value). Retrieving the context size or the digest size requires the hash:

```
SceLibSecureErrorType error;
size_t context_size, digest_size;

/* Retrieve the context size */
error = sceLibSecureHashGetContextSize(SCE_LIBSECURE_HASH_SHA1,
     &context_size);
if(error != SCE_LIBSECURE_OK)
{
    /* Error management */
    ...
}

/* Retrieve the digest size */
error = sceLibSecureHashGetDigestSize(SCE_LIBSECURE_HASH_SHA1,
     &digest_size);
if(error != SCE_LIBSECURE_OK)
{
    /* Error management */
    ...
}
```

(2) Set the context according to the parameters passed when retrieving the context size, and allocate the memory for the HMAC value:

```
SceLibSecureErrorType error;
SceLibSecureBlock context;
SceLibSecureHmac hmac;

/* Allocate the context */
context.length = context_size;
context.pointer = malloc(context.length);
/* Allocate the HMAC */
hmac.length = digest_size;
hmac.pointer = malloc(hmac.length);
```

```
if(context.pointer != NULL && hmac.pointer != NULL)
{
    /* Set the context */
    error = sceLibSecureHashSetContext(&context,
       SCE_LIBSECURE_HASH_SHA1);
    ...
}
```

Authenticating the Message

Finally, to process the message you want to authenticate, use the `sceLibSecureHashHmac` function. This function requires the context, the key (`SceLibSecureSymmetricKey` for symmetric ciphers; `SceLibSecureAsymmetricKey` for asymmetric ciphers), and the message.

```
SceLibSecureErrorType error;

/* Authenticate the message */
error = sceLibSecureHashHmac(&context, &hmac, &key, &message);
if(error != SCE_LIBSECURE_OK)
{
    /* Error management */
    ...
}
```

Destroying the Context

It is good practice to destroy the previously used context to prevent the data being used or if you do not plan to reuse the same context to continue the encryption or the decryption process.

To destroy the context, use the `sceLibSecureHashDeleteContext` function:

```
SceLibSecureErrorType error;

/* Destroy the context */
error = sceLibSecureHashDeleteContext(&context);
...
```

Library Termination

To fully terminate the library, call the `sceLibSecureDestroy` function. This function ensures that the library has cleared the memory used.

```
/* Destroy the library */
error = sceLibSecureDestroy();
...
```