Crypto Manual

Software Encryption Library

2008/09/16

TWL-06-0006-001-A
Released: October 20, 2008

Table of Contents

Figures

Revision History

Revision Date	Description
2008/09/16	Initial version.

1 Introduction

1.1 Overview

The Crypto library is used for encryption. By encrypting data used in a game it is possible to hide data and use digital signatures to authenticate, which will detect tampering. The TWL-SDK library contains another encryption library: the AES library. However, the AES library is used for the AES encryption feature that is in TWL hardware. The Crypto library provides software encryption using types of encryption other than AES.

Note: To export this product from Japan, you must obtain permission by applying to the Ministry of Economy, Trade and Industry for permission to export. Permission is required due to the fact that the Crypto library includes built-in features that function as an encryption device as defined under the Foreign Exchange and Trade Law and Export and Trade Control Act of Japan. Use proper care and caution when using this library. The export regulations and laws of each nation involved must be observed when exporting this library package.

1.2 CRYPTO Library Structure

The Crypto library consists of the following modules.

- RC4 Encryption: Encryption and decryption are performed using an RC4 private key.
- Digital Signatures: Performs digital signature verification. It is also possible to create a signature on TWL.
- RSA encryption: Provides encryption and decryption using a public key code.
- UTIL: Provides the functions that are required to operate the Crypto library, but that are not provided in the other modules.

1.3 Characteristics of Each Encryption Method

The TWL-SDK provides the following encryption algorithms (ciphers) in the Crypto library RC4, RSA; in the AES library AES.

Compared to other ciphers provided in the TWL-SDK, RC4 is faster, and encryption and decryption can be done with the same routine, making it easy to use. However, the same key is used for encryption and decryption. If the key were disclosed, users could easily decipher, which defeats the purpose of encryption. Therefore, RC4 cannot be used alone to encrypt highly confidential data. RC4 has existed since Nitro-Crypto, so use RC4 for applications that include data transfer on Nitro.

AES is a stronger cipher than RC4. We recommend using AES to encrypt data that will only be used in TWL. However, take care to avoid disclosing the key because AES also uses the same key for encryption and decryption.

The RSA cipher uses a public key method. Since disclosing a public key is not a risk, key management is significantly safer than shared key methods such as RC4 and AES. Also, RSA key length in the Crypto library can be up to 4096 bits, making encryption stronger than that of other encryption methods. However, processing speed is much slower. Therefore the normal method is to limit using RSA to encrypt only the AES key, for which there is a key transmission risk, and to encrypt the data with AES.

1.4 Operating Environment

RC4, digital signature verification, and UTIL run in TWL and Nitro.

RSA and digital signature creation do not run in Nitro due to licensing restrictions on the encryption core that they use. They also do not run on Nitro even for Hybrid applications.

2 RC4 Encryption

2.1 Purpose and Restrictions

Encryption functions that use the Crypto library RC4 cipher have been prepared to work with both Nitro and TWL. We assume that they will be used for data that must also be handled by Nitro, such as when it is sent over a network. We recommend AES when it is not necessary for Nitro to handle the data.

However, because shared key encryption is used with RC4, the key data used for both encryption and decryption must be stored in the software. Therefore, if the key is discovered through analyzing the ROM binary, encryption will be threatened. **Do not use this function alone for encrypting highly sensitive data or for verifying the data's author.**

2.2 Features of the RC Algorithm

The RC4 algorithm has the following features.

- Public key encryption
- Stream encryption
- High-speed encryption/decryption
- An efficient analysis technique has not been announced

Using stream encryption is simple because the number of input bytes matches the number of output bytes. However, it may not remain robust if certain precautions are not followed. Be sure to note the precautions described below.

2.3 Principle of Operation of the RC Algorithm

The RC4 algorithm works by creating a uniquely defined random number string from the key, then XOR-ing the original data with the random number string. As a result, the same key always generates the same encryption random number string. This is described as follows:

1. The same encrypted data is always generated given the same key and the same source data. It is possible to know when two sets of encrypted text represent the same plain text (dictionary attack).
2. When two sets of encrypted data generated using the same encryption key are XOR-ed, the same result is obtained as when the original plain text data is XOR-ed (one type of differential attack).
3. Reversing one bit of data anywhere in the encrypted text will result in the reversal of one bit of data after data is decrypted (bit inversion attack).

In order to foil the dictionary attacks and differential attacks, a unique initialization vector (IV) is created each time and added to the public key to create a real key for RC4 algorithm to use. When the encrypted data is sent, the unencrypted IV must also be sent. For example, out of the 128 bits passed as a key to the RC4 function, 96 are handled as a true private key, while the remaining 32 are filled with a different number each time the function is invoked for use as the initialization vector.

To avoid bit inversion attacks, a message digest value such as MD5 or SHA-1 is attached to any data to be sent. Because an attacker does not know the original data, he or she cannot calculate the correct message digest value even if he or she manages to change any bits. Functions for finding MD5 and SHA-1 are provided in the NITRO-SDK.

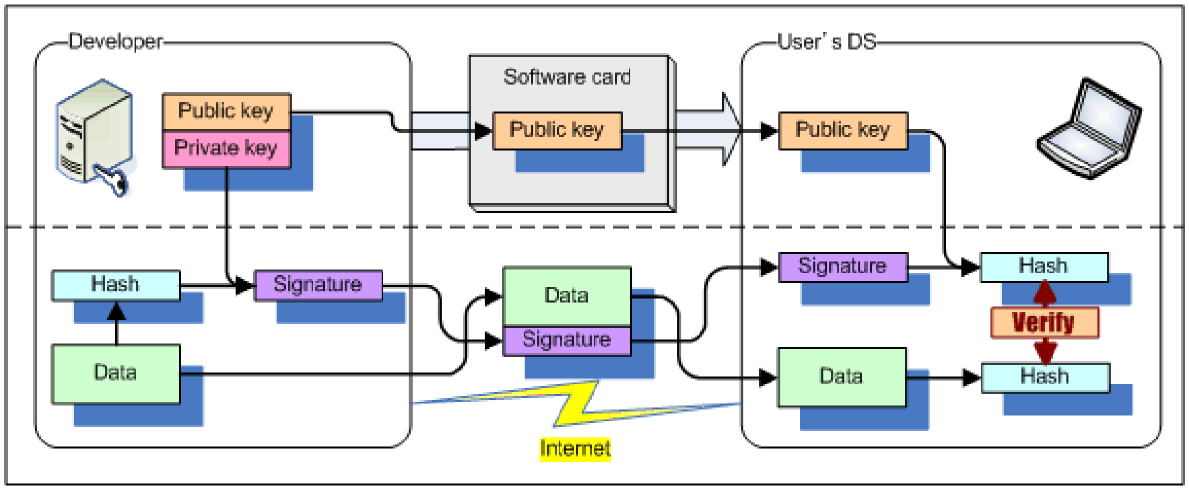
For more information, refer to any basic text on encryption technology.

3 Digital Signatures

3.1 What Is a Digital Signature?

A digital signature is a mechanism that is used to verify the authenticity of data that has been received via an untrusted route such as the Internet.

Figure 3-1 Digital Signature Block Diagram



- The signing party (the developer) and the verifying party (the user's DS) participate as working subjects.
- The data involved consists of the private key, public key, send data, and digital signature for the send data.
- The private key is a secret known only to the signing party.
- A public key must be obtained ahead of time by the verifying party using some reliable means (such as having it burned onto the game card).
- The send data is a binary file of any size.

The process flow used to verify the integrity of data when using digital signatures is as follows.

1. The signer uses the private key to create a fixed-length digital signature from the send data. (Actually, a hash value for the send data is obtained and operations are performed on that.)
2. The verifying party receives the send data and the digital signature for that data via the Internet or other means.
3. The verifying party can verify the authenticity of the data based on the send data and the signature using only the public key information obtained ahead of time.

Digital signatures have the following features.

- As long as the verifying party has the public key ahead of time, he or she can determine authenticity with only the data and the signature associated with it without any further external communications.
- As long as the private key is not leaked, a signature cannot be forged even if the public key is known. (In other words, signatures cannot be forged even if the ROM binary on the DS is somehow analyzed.)

3.2 Digital Signature Features Provided by the CRYPTO Library

The two main digital signature features are as follows.

- Verify digital signatures on the system (Nitro and TWL)
- Create digital signatures on the system (only TWL)

The `CRYPTO_VerifySignature` and `CRYPTO_VerifySignatureWithHash` functions are provided for digital signature verification. These have been in existence since Nitro-Crypto, so they can also be used with Nitro.

`CRYPTO_RSA_Sign` and other functions are provided for creating digital signatures. This feature uses RSA Security's BSAFE Micro Edition Suite and for licensing reasons can only be used in TWL.

The Crypto library does not have certificate management features such as certificate expiration date management. If you need such features, implement them from the application.

A digital signature only verifies the legitimacy of the data. Data encryption is not performed. We have provided RC4 and RSA encryption in the Crypto library, and the AES library, so you may use any of them depending on the encryption strength that you need. If you want to have safe communication with a server using the TwlWiFi library, use the NSSL library (SSL communication library).

3.3 Signature Data Format

Signature data passed to the `CRYPTO_VerifySignature*` function can be created according to any method as long as it satisfies the conditions listed below.

- Conforms to PKCS#1
- Uses SHA-1 as the hash algorithm.
- Uses RSA for the public key encryption algorithm with a key length of 1024 bits.
- The public exponent of the public key is 65537.

3.4 Examples of Creating Signature Data

Digital signatures can also be created on a TWL system. The following is an example of the procedure for generating a digital signature using OpenSSL, the open source SSL toolkit.

3.4.1 Generating an RSA Key for the Signature

Input the following commands on the command line when OpenSSL is installed to generate a 1024-bit long RSA key file, `privkey.pem`.

```
> openssl genrsa -out privkey.pem 1024
```

If this file were leaked, anyone would be able to sign data with it. The private key file therefore needs to be maintained with the strictest care.

Password-based encryption for `privkey.pem` is possible by specifying the encryption method when generating the key. In the following example, a newly generated `privkey.pem` file is encrypted with the 3DES algorithm.

```
> openssl genrsa -des3 -out privkey.pem 1024
```

For more information, see the OpenSSL reference material.

3.4.2 Checking the Contents of an RSA Key

Use the following command to confirm the content of `privkey.pem`.

```
> openssl rsa -in privkey.pem -text -noout
```

It includes the private information needed for signing, but there are also two values needed for verification (via the public key): `modulus` and `publicExponent`.

The following is an example of extracted `modulus` and `publicExponent` values output from the command.

```
modulus:
00:eb:95:be:33:19:73:64:f2:72:2c:87:c8:0a:f3:
1c:ba:e0:4c:e0:3e:1d:f6:e2:09:aa:70:f0:b3:b9:
0c:86:36:62:2d:12:13:86:fa:a5:3d:93:cb:5f:0b:
45:64:9b:7b:eb:b5:c6:f9:42:99:70:46:f3:14:6d:
8f:f9:b9:ec:38:30:a0:1c:28:0d:30:d9:86:1a:4d:
1b:f2:e9:05:1b:43:06:b2:c0:55:ed:c4:bb:8e:1a:
a5:ab:2b:54:e5:dc:8d:70:cf:af:91:94:c9:e9:8f:
7f:9f:29:28:be:e7:01:b0:20:d4:f2:71:58:93:db:
25:1c:26:bc:98:f3:a2:b3:47
publicExponent: 65537 (0x10001)
```

Since the public exponent used by the `CRYPTO_VerifySignature*` function is fixed at 65537, confirm that the `publicExponent` has a value of 65537.

The `modulus` value can also be generated with the following command.

```
> openssl rsa -in privkey.pem -modulus -noout
```

This command outputs a text string similar to the following.

```
Modulus=EB95BE33197364F2722C87C80AF31CBAE04CE03E1DF6E209AA70F0B3B90C8636622D121386F
AA53D93CB5F0B45649B7BEBB5C6F942997046F3146D8FF9B9EC3830A01C280D30D9861A4D1BF2E9051B
4306B2C055EDC4BB8E1AA5AB2B54E5DC8D70CFAF9194C9E98F7F9F2928BEE701B020D4F2715893DB251
C26BC98F3A2B347
```

The hexadecimal value following "`Modulus=`" must be converted to an u8 array in C and passed as `mod_ptr` to the `CRYPTO_VerifySignature*` function. In the example above, the modulus is 127 bytes long because the leading zeros are omitted when the length is less than 128 bytes. When passing the value to `mod_ptr`, be sure to restore any leading zeros to maintain a length of 128 bytes.

3.4.3 Creating a Digital Signature

Once the above steps have been completed, all that remains is to create a digital signature for the target data.

Digital signatures can be created on the TWL using `CRYPTO_RSA_Sign`, and can also be created on PC.

The following command uses the `privkey.pem` private key to generate the signature data, `hoge.sign`, used to sign `hoge.txt`.

```
> openssl dgst -sha1 -sign privkey.pem -out hoge.sign hoge.txt
```

Confirm that the resulting file size is 128 bytes.

This 128-byte binary data is transferred to a DS and passed to the `CRYPTO_VerifySignature*` function as `sign_ptr`.

Run the following command to confirm on the PC whether the generated signed data forms a valid digital signature.

```
> openssl dgst -sha1 -prverify privkey.pem -signature hoge.sign hoge.txt
```

3.4.4 Verifying the Digital Signature

By embedding public key data in a DS program in advance, it can then receive data and digital signature data. Providing the `CRYPTO_VerifySignature` function with the data, the data size, the digital signature data (128 bytes), and the embedded public key data (the 128-byte modulus) allows the digital signature to be verified. This function returns a value of TRUE if the data is determined to be valid.

4 RSA Encryption

4.1 About RSA Encryption

The encryption functions that use the RSA algorithm were prepared for encryption using public keys.

The RSA algorithm has the following characteristics.

- It is a type of public-key encryption.
- Encryption is stronger than the RC4 or AES ciphers that can be used in the TWL-SDK.
- The processing for encryption and decryption is extremely slow.

One advantage of public-key cryptography is that, compared to shared-key cryptography, the risk associated with key transmission is low. A disadvantage is that the processing speed is slow relative to some other cryptographic methods. As a result, it is common to encrypt the data you want to encrypt using some algorithm other than RSA, then use RSA to encrypt the key that was used for that encryption and send that.

4.2 Precautions About the Use of RSA Encryption

The RSA encryption algorithm has the following properties.

- The encryption is completely circumvented if the private key is compromised.
- Decryption is possible if one can identify the private exponent (brute-force attack).
- It is possible to spoof data if one were to abuse the key transmission (man-in-the-middle attack).

If the private key is compromised, it is possible both to defeat the encryption and to falsify signatures, and the safety provided by the encryption is lost. You must therefore be careful about how you manage your private keys.

Increasing the key length makes it easy to defend against brute-force attacks, but the longer you make your keys, the slower the encryption will become.

Verification (signatures) of public keys is useful for preventing man-in-the-middle attacks.

For more information, refer to any basic text on encryption technology.

4.3 Key Format and Encryption/Decryption Strings

The DER format is used for both public and private keys. There is no limit on the length of the keys.

The encryption strings must be shorter than the key length. (For example, if the key length is 1024 bits, the string to encrypt must be less than 1024 bits.) If the strings to decrypt were not encrypted with the Crypto library, they must meet all of the following requirements.

- It conforms to PKCS #1.
- The public exponent of the public key is 65537.

4.4 Limitations

Due to software licensing restrictions, the RSA encryption algorithm can only be used in TWL. It cannot be used in Nitro. When used in Hybrid applications, it will only function in TWL.

4.5 Key Creation Example

The following is an example of the procedure for generating public and private keys for encryption with OpenSSL from the open source SSL toolkit.

4.5.1 Create an RSA Private Key

Input the following commands in a command line on a system on which OpenSSL has been installed. This will generate a 1024-bit-long RSA private key file in PEM format, `privkey.pem`.

```
> openssl genrsa -out privkey.pem
```

In the event that `privkey.pem` were to be leaked or compromised, anyone would be able to break or falsify the encryption. The private key file therefore needs to be maintained with the strictest care.

Once the private key has been created in PEM format, convert it to DER format.

```
> openssl rsa -outform DER -in privkey.pem -out privkey.der
```

When specifying a private key with the CRYPTO API, convert the content of this `privkey.der` file to a C-language u8 array. The `privkey.der` file is a private key just like `privkey.pem`, so it should be handled in an equally strict manner.

4.5.2 Create an RSA Public Key

Create a public key in DER format with the following command.

```
> openssl rsa -pubout -inform DER -in privkey.der -outform DER -out pubkey.der
```

When specifying a public key with the CRYPTO API, convert the content of this `pubkey.der` file to a C-language u8 array.

4.5.3 Check the Functionality of the Keys

Make sure the pair of private and public keys you generated is functioning properly.

1. First, prepare a text file (`test.txt`) that contains a string that is shorter than the key, and encrypt it using the public key, converting it to `test.txt.enc`.

```
> openssl rsautl -encrypt -in test.txt -out test.txt.enc -pubin -keyform DER -inkey pubkey.der
```

   Make sure the pair of private and public keys you generated is functioning properly.

2. Next, decode `test.txt.enc` using the private key, converting it to `test.txt.dec`.

```
> openssl rsautl -decrypt -in test.txt.enc -out test.txt.dec -keyform DER -inkey privkey.der
```

   If the content of `test.txt` matches that of `test.txt.dec`, you have confirmed that the keys are functioning properly.