PKCS #11 Message-Based Encryption and Decryption

Wan-Teh Chang <wtc@google.com>, 2014-03-25

Introduction

Message-based encryption refers to the process of encrypting multiple messages using the same encryption mechanism and encryption key. The encryption mechanism can be an authenticated encryption with associated data (AEAD) algorithm but can also be a pure encryption algorithm.

The main goal of this proposal is to make this process as efficient as possible by reducing the number of PKCS #11 calls required. A second goal is to allow a crypto token to generate the initialization vector (IV) or nonce for the encryption mechanism.

This proposal incorporates ideas, terminology, and proposals of Michael StJohns and Bob Relyea.

Proposal

I first specify the new functions, and then specify an AES-GCM mechanism for message-based encryption as a concrete example.

Design issues, my recommended resolutions, and alternative designs are covered in the **Notes** after the specification.

CK_CONST_BYTE_PTR and CK_CONST_VOID_PTR

To convey the fact that a function won't modify an input byte array, this proposal uses the `CK_CONST_BYTE_PTR` type, which isn't defined in v2.40:

```
typedef const CK_BYTE CK_PTR CK_CONST_BYTE_PTR;
```

Similarly this proposal uses `CK_CONST_VOID_PTR` to convey the fact that a function won't modify a generic input parameter:

```
typedef const void CK_PTR CK_CONST_VOID_PTR;
```

C_EncryptAssociationInit

```
CK_RV C_EncryptAssociationInit(
  CK_SESSION_HANDLE hSession,
  CK_MECHANISM_PTR pMechanism,
  CK_OBJECT_HANDLE hKey
);
```

The **C_EncryptAssociationInit** function prepares a session for one or more encryption operations that use the same encryption mechanism and encryption key. We use the term ***message-based encryption*** to refer to the process of encrypting multiple messages using the same mechanism and key.

After calling **C_EncryptAssociationInit**, an application can call **C_EncryptMessage** to encrypt a message in a single part. This may be repeated several times. Finally, an application calls **C_EncryptAssociationFinal** to finish the message-based encryption process.

An encryption mechanism may request that the crypto token generates the initialization vectors (IVs) or nonces used in subsequent encryption operations by specifying an IV generator mechanism in the encryption mechanism's parameter.

Notes:

1. **C_EncryptAssociationInit** creates an *encryption association* object that holds the encryption mechanism and encryption key common to the subsequent encryption operations. In keeping with the style of existing PKCS #11 functions, this proposal makes the encryption association object implicit in the session. An alternative design is to return an encryption association object handle in a CK_OBJECT_HANDLE_PTR output argument and pass that handle (instead of the session handle) to the message encryption functions.
2. **C_EncryptAssociationInit** has the same function prototype as **C_EncryptInit**. An alternative design is to simply use **C_EncryptInit** to start a message-based encryption process, and let each mechanism specify whether it should be used with the **C_Encrypt**/**C_EncryptUpdate** functions or the new **C_EncryptMessage** function. A new variant of **C_EncryptFinal** that takes just a session handle input is still needed to finish a message-based encryption process.
3. This proposal only specifies a **C_EncryptMessage** function that encrypts a message in a single part. If the need arises, we can extend the proposal with additional functions for encrypting a message in multiple parts.
4. By using additional function parameters, it is possible to support both single-part and multiple-part operations using just one or two functions. The versatility of such functions comes at the expense of code readability. In keeping with the style of existing PKCS #11 functions, I am in favor of specifying specialized functions for single-part and multiple-part

operations.

5. The encryption mechanism's parameter may specify an IV generator *mechanism*. It is possible to specify an IV generator *object handle* instead. This alternative design requires new functions to create, use, and destroy an IV generator, and problems of two message-based encryption processes (possible using different keys) sharing the same IV generator incorrectly could arise. This proposal aims for easy of use and lets the encryption association manage and use an IV generator internally.
6. The IV generator mechanism could also be a separate input parameter of `C_EncryptAssociationInit`. This would allow an IV generator to be used with an existing mechanism such as CKM_AES_CBC without defining a new mechanism parameters structure.

C_EncryptMessage

```
CK_RV C_EncryptMessage(
  CK_SESSION_HANDLE hSession,
  CK_VOID_PTR pParameter,
  CK_ULONG ulParameterLen,
  CK_CONST_BYTE_PTR pAssociatedData,
  CK_ULONG ulAssociatedDataLen,
  CK_CONST_BYTE_PTR pPlaintext,
  CK_ULONG ulPlaintextLen,
  CK_BYTE_PTR pCiphertext,
  CK_ULONG_PTR pulCiphertextLen
);
```

The **C_EncryptMessage** function encrypts a message in a single part. It does not finish the message-based encryption process. Additional **C_EncryptMessage** calls may be made on the session.

`pParameter` and `ulParameterLen` specify any mechanism-specific parameters for the encryption operation. Typically this is an initialization vector (IV) or nonce. Depending on the mechanism parameter passed to **C_EncryptAssociationInit**, `pParameter` may be either an input or an output parameter. For example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV generator will be output to the `pParameter` buffer.

`pAssociatedData` and `ulAssociatedDataLen` specify the associated data for an AEAD mechanism. If the mechanism is not AEAD, `pAssociatedData` and `ulAssociatedDataLen` are not used and should be set to (NULL, 0).

Notes:

1. Some AEAD algorithms such as AES-GCM produce a separable authentication tag. Making the authentication tag available as a separate output parameter is more flexible and may save the caller the need to copy data. On the other hand, it is conceivable that an AEAD algorithm can "mix in" the integrity check throughout the ciphertext, and therefore can't use the authentication tag output. This proposal follows the AEAD API described in RFC 5116 and uses a single output (the ciphertext). I also consulted the requirements for the competition for authenticated encryption CAESAR, and found that it defines an authenticated cipher as "a function with five byte-string inputs and one byte-string output. … The output is a variable-length ciphertext." To support this design decision, I surveyed the commonly used security protocols (TLS, IPsec, SSH, S/MIME, and Kerberos). Only one of them, S/MIME, must use a separate MAC field (see RFC 5083 and RFC 5084). Since both TLS and IPsec were successfully revised to support AEAD algorithms (see RFC 5246 Section 6.2.3.3 and RFC 4303 Section 3.2.3), I am hopeful that S/MIME (or rather CMS) can also be updated to better support AEAD algorithms.
2. As noted above, `pAssociatedData` and `ulAssociatedDataLen` are not used for non-AEAD algorithms. Although not ideal, this is an acceptable trade-off that allows us to use the same function for both AEAD and non-AEAD algorithms.
3. All the AEAD algorithms I surveyed require the entire associated data to be available before the first byte of plaintext is given to the encryption function. Therefore it doesn't seem necessary to support both an associated data *prefix* and an associated data *suffix* input.

C_EncryptAssociationFinal

```
CK_RV C_EncryptAssociationFinal(
  CK_SESSION_HANDLE hSession
);
```

The **C_EncryptAssociationFinal** function finishes a message-based encryption process initiated by an earlier **C_EncryptAssociationInit** call.

C_DecryptAssociationInit

```
CK_RV C_DecryptAssociationInit(
  CK_SESSION_HANDLE hSession,
  CK_MECHANISM_PTR pMechanism,
  CK_OBJECT_HANDLE hKey
);
```

The **C_DecryptAssociationInit** function prepares a session for one or more decryption

operations that use the same decryption mechanism and decryption key. We use the term ***message-based decryption*** to refer to the process of decrypting multiple messages using the same mechanism and key.

After calling **C_DecryptAssociationInit**, an application can call **C_DecryptMessage** to decrypt a message in a single part. This may be repeated several times. Finally, an application calls **C_DecryptAssociationFinal** to finish the message-based decryption process.

In subsequent decryption operations, IVs are provided by the caller, so the decryption mechanism's parameter must not specify an IV generator mechanism.

Notes:

1. This proposal only specifies a **C_DecryptMessage** function that decrypts a message in a single part. If the need arises, we can extend the proposal with additional functions for decrypting a message in multiple parts. However, it is recommended or even mandated that AEAD decryption should not release any plaintext if the authenticity of the input cannot be verified. Therefore, each decryption mechanism must specify whether or not it can be used with the multiple-part message decryption functions.

C_DecryptMessage

```
CK_RV C_DecryptMessage(
  CK_SESSION_HANDLE hSession,
  CK_CONST_VOID_PTR pParameter,
  CK_ULONG ulParameterLen,
  CK_CONST_BYTE_PTR pAssociatedData,
  CK_ULONG ulAssociatedDataLen,
  CK_CONST_BYTE_PTR pCiphertext,
  CK_ULONG ulCiphertextLen,
  CK_BYTE_PTR pPlaintext,
  CK_ULONG_PTR pulPlaintextLen
);
```

The **C_DecryptMessage** function decrypts a message in a single part. It does not finish the message-based decryption process. Additional **C_DecryptMessage** calls may be made on the session.

`pParameter` and `ulParameterLen` specify any mechanism-specific parameters for the decryption operation. Typically this is an initialization vector (IV) or nonce. Unlike the same-named parameter of **C_EncryptAssociationInit**, `pParameter` is always an input parameter.

pAssociatedData and ulAssociatedDataLen specify the associated data for an AEAD mechanism. If the mechanism is not AEAD, pAssociatedData and ulAssociatedDataLen are not used and should be set to (NULL, 0).

If the decryption mechanism is an AEAD algorithm and the authenticity of the associated data or ciphertext cannot be verified, **C_DecryptMessage** returns the error code CKR_AEAD_DECRYPT_FAILED.

```
#define CKR_AEAD_DECRYPT_FAILED 0x00000042
```

Notes:

1. CKR_AEAD_DECRYPT_FAILED is a new error code. Alternatively, we can use the existing error code CKR_ENCRYPTED_DATA_INVALID (0x00000040) for this purpose.

C_DecryptAssociationFinal

```
CK_RV C_DecryptAssociationFinal(
  CK_SESSION_HANDLE hSession
);
```

The **C_DecryptAssociationFinal** function finishes a message-based decryption process initiated by an earlier **C_DecryptAssociationInit** call.

AES-GCM Mechanism for Message-Based Encryption

This section specifies a new AES-GCM mechanism for message based encryption.

CKM_AES_GCM_V2

```
#define CKM_AES_GCM_V2 0x00000700
```

CKM_AES_GCM_V2 is an AES-GCM mechanism for message-based encryption and decryption. It only supports single-part message encryption and decryption. It must not be used with multiple-part message encryption and decryption functions.

CK_GCM_PARAMS_V2

The mechanism parameter structure for CKM_AES_GCM_V2 is CK_GCM_PARAMS_V2. It is not AES-specific and therefore can also be used with Camellia-GCM.

```
typedef struct CK_GCM_PARAMS_V2 {
  CK_ULONG ulIvLen;  /* Length in bytes of the IV */
  CK_ULONG ulTagLen;  /* Length in bytes of the authentication tag */
  CK_MECHANISM_PTR pIvGeneratorMech;  /* IV generator mechanism */
} CK_GCM_PARAMS_V2;
```

The `pIvGeneratorMech` field specifies the IV generator mechanism for encryption.

- If NULL, the IV is provided by the caller. The `pParameter` parameter of **C_EncryptMessage** is an input.
- If not NULL, it points to a `CK_MECHANISM` structure that specifies how the IV is generated partially or fully by the crypto token. The `pParameter` parameter of **C_EncryptMessage** is an output.

For decryption, the `pIvGeneratorMech` field is not used and must be set to NULL.

Notes:

1. When the IV is provided by the caller, the `pIvGeneratorMech` field could be used to instruct the crypto token how it should validate the externally-provided IV. For example, it can tell the crypto token that the provided IV should be a monotonically increasing big-endian integer.

IV Generator Mechanisms

This section specifies two representative IV generator mechanisms.

CKM_IV_GEN_DETERMINISTIC

```
#define CKM_IV_GEN_DETERMINISTIC 0x00000750
```

`CKM_IV_GEN_DETERMINISTIC` is an IV generator mechanism specified in NIST SP 800-38D, Sec. 8.2.1 Deterministic Construction. The fixed field is on the left and the invocation field is on the right. The caller provides the fixed field. The crypto token generates and outputs the invocation field.

CK_IV_GEN_DETERMINISTIC_PARAMS

The mechanism parameter structure for `CKM_IV_GEN_DETERMINISTIC` is the `CK_IV_GEN_DETERMINISTIC_PARAMS` structure:

```
typedef struct CK_IV_GEN_DETERMINISTIC_PARAMS {
  CK_CONST_BYTE_PTR pFixed;  /* the fixed field */
```

```
  CK_ULONG ulFixedLen;
} CK_IV_GEN_DETERMINISTIC_PARAMS;
```

In each **C_EncryptMessage** call, `pParameter` should point to an output buffer receiving the invocation field generated by the crypto token. `ulParameterLen` is the length of that output buffer and should be `blocksize - ulFixedLen`, where `blocksize` is the symmetric cipher's block size (16 for AES).

CKM_IV_GEN_RGB_BASED

```
#define CKM_IV_GEN_RGB_BASED 0x00000751
```

`CKM_IV_GEN_RGB_BASED` is an IV generator mechanism specified in NIST SP 800-38D, Sec. 8.2.2 RGB-based Construction. The random field is on the left and the free field is on the right. The caller provides the free field. The crypto token generates and outputs the random field. The free field is recommended to be empty.

CK_IV_GEN_RGB_BASED_PARAMS

The mechanism parameter structure for `CKM_IV_GEN_RGB_BASED` is the `CK_IV_GEN_RGB_BASED_PARAMS` structure:

```
typedef struct CK_IV_GEN_RGB_BASED_PARAMS {
  CK_CONST_BYTE_PTR pFree;  /* the free field */
  CK_ULONG ulFreeLen;
} CK_IV_GEN_RGB_BASED_PARAMS;
```

In each **C_EncryptMessage** call, `pParameter` should point to an output buffer receiving the random field generated by the crypto token. `ulParameterLen` is the length of that output buffer and should be `blocksize - ulFreeLen`, where `blocksize` is the symmetric cipher's block size (16 for AES).

Appendix: Multiple-Part Message-Based Encryption Functions

This appendix outlines two functions for encrypting a message in multiple parts.

Given the single-part message-based encryption function specified above:

```
CK_RV C_EncryptMessage(
  CK_SESSION_HANDLE hSession,
  CK_VOID_PTR pParameter,
```

```
  CK_ULONG ulParameterLen,
  CK_CONST_BYTE_PTR pAssociatedData,
  CK_ULONG ulAssociatedDataLen,
  CK_CONST_BYTE_PTR pPlaintext,
  CK_ULONG ulPlaintextLen,
  CK_BYTE_PTR pCiphertext,
  CK_ULONG_PTR pulCiphertextLen
);
```

we can split it into two functions:

```
CK_RV C_EncryptMessageBegin(
  CK_SESSION_HANDLE hSession,
  CK_VOID_PTR pParameter,
  CK_ULONG ulParameterLen,
  CK_CONST_BYTE_PTR pAssociatedData,
  CK_ULONG ulAssociatedDataLen
);

CK_RV C_EncryptMessageNext(
  CK_SESSION_HANDLE hSession,
  CK_CONST_BYTE_PTR pPlaintextPart,
  CK_ULONG ulPlaintextPartLen,
  CK_BYTE_PTR pCiphertextPart,
  CK_ULONG_PTR pulCiphertextPartLen,
  CK_FLAGS flags
);

#define CKF_END_OF_MESSAGE 0x00000001
```

The `flags` argument is set to 0 if there is more plaintext data to follow, or set to `CKF_END_OF_MESSAGE` if this is the last plaintext part.

Alternative designs are possible. For example, in the following alternative design, **C_EncryptMessageBegin** may process plaintext data, and may even operate in single-part mode if the `CKF_END_OF_MESSAGE` flag is specified, eliminating the need for the specialized single-part function **C_EncryptMessage**:

```
CK_RV C_EncryptMessageBegin(
  CK_SESSION_HANDLE hSession,
  CK_VOID_PTR pParameter,
  CK_ULONG ulParameterLen,
  CK_CONST_BYTE_PTR pAssociatedData,
```

```
  CK_ULONG ulAssociatedDataLen,
  CK_CONST_BYTE_PTR pPlaintext,
  CK_ULONG ulPlaintextLen,
  CK_BYTE_PTR pCiphertext,
  CK_ULONG_PTR pulCiphertextLen,
  CK_FLAGS flags
);

CK_RV C_EncryptMessageNext(
  CK_SESSION_HANDLE hSession,
  CK_CONST_BYTE_PTR pPlaintextPart,
  CK_ULONG ulPlaintextPartLen,
  CK_BYTE_PTR pCiphertextPart,
  CK_ULONG_PTR pulCiphertextPartLen,
  CK_FLAGS flags
);

#define CKF_END_OF_MESSAGE 0x00000001
```

The **C_EncryptMessageNext** function in this design is syntactic sugar, so this can be further pared down to just one function. I don't like a function that requires the caller to pass more than one (NULL, 0) because the code will be less readable. So I think the syntactic sugar is worthwhile.