Amendments Proposals to PKCS#11

for support of

Secondary PIN handling and WTLS

This document extends

Title	Document No
PKCS#11 v2.11: Cryptographic Token Interface Standard	RSA Laboratories November 2001 http://www.rsasecurity.com/rsalabs/PKCS/pkcs-11/index.html

TABLE OF CONTENTS:

1 Introduction

This document contains proposals for amendments to the PKCS#11 Cryptographic Token Interface Standard. The purpose of these amendments is to provide support through a standardized PKCS#11 interface for better secondary PIN handling and to provide support for WTLS. We also propose an amendment for TLS support. New data types and mechanisms are described.

The current official (and draft) version describes a solution for the secondary PIN handling which is complex. The solution that is described here is simple yet adequate to handle secondary (or non-repudiation) PINs on PKCS#15 based tokens.

In addition we suggest a standardized way to support WTLS a TLS derived transport security layer that is used in WAP environments.

1.1 Terminology

Definition/Abbreviation	Explanation
IV	Initialization vector
PKCS	Public-Key Cryptography Standards
PRF	Pseudo random function
RSA	The RSA public key crypto system
SW	Software
TLS	Transport Layer Security
WIM	Wireless Identification Module
WTLS	Wireless Transport Layer Security

1.2 Revision History

Revisions	Date	Changes since the former revision
A	2002-09-26 Matthias Esswein	First version.

1.3 References

No	Title	Document No
1	PKCS#11 v2.11: Cryptographic Token Interface Standard	RSA Laboratories November 2001 http://www.rsasecurity.com/rsalabs/PKCS/pkcs-11/index.html
2	Wireless Transport Layer Security Version 06-Apr-2001	Wireless Application Protocol WAP-261-WTLS-20010406-a http://www.wapforum.org/
3	The TLS Protocol Version 1.0	RFC 2246 The Internet Engineering Task Force, January 1999 http://www.ietf.org/

1.4 Yet to do

-

2 Modified general overview

This chapter contains additions to Chapter 6 of [1].

2.1 Secondary authentication

This chapter replaces Chapter 6.7 of [1].

Cryptoki allows an application to specify that a private key should be protected by a secondary authentication mechanism. This mechanism is additional to the standard login mechanism for sessions described in Chapter 6.6 of [1].

The intent of secondary authentication is to provide a means for a cryptographic token to produce digital signatures for non-repudiation with reasonable certainty that only the authorized user could have produced that signature. This capability is becoming increasingly important as digital signature laws are introduced worldwide.

The secondary authentication is based on the following principles:

- The owner of the private key must be authenticated to the token before secondary authentication can proceed (i.e. **C_Login** must have been called successfully).
- If a private key is protected by a secondary authentication PIN, then the token must require that the PIN be presented before each use of the key for any purpose.

The secondary authentication mechanism adds a couple of subtle points to the way that an application presents an object to a user and generates new private keys with the additional protections. The following sections detail the minor additions to applications that are required to take full advantage of secondary authentication.

2.1.1 Using keys protected by secondary authentication

Using a private key protected by secondary authentication uses a slightly modified process, and call sequence, as using a private key that is only protected by the login PIN.

When a cryptographic operation, such as a digital signature, is started using a key protected by secondary authentication, the required PIN value will be gathered by means of the function **C_AuthenticateObject** described in Chapter 5.2.1. If the PIN is correct, then the operation is allowed to complete. Otherwise, the function will return an appropriate error code.

The application can detect when Cryptoki and the token will gather a PIN for secondary authentication by querying the key for the **CKA_SECONDARY_AUTH** attribute (see Chapter 4.1). If the attribute value is TRUE, then the application can present a prompt to the user. The second possibility is that the operation using a private key protected by secondary authentication will fail with the new return value **CKR_SECONDARY_AUTHENTICATION_REQUIRED** if the PIN wasn't verified for this usage. So this value can trigger the application to prompt for the required PIN.

2.1.2 Generating private keys protected by secondary authentication

To generate a private key protected by secondary authentication, the application supplies the **CKA_SECONDARY_AUTH** attribute with value TRUE and the **CKA_AUTH_PIN** and **CKA_AUTH_PIN_LEN** attributes in the private key template. If the attributes do not exist in the template or has the value FALSE, then the private key is generated with the normal login protection. See Chapter 11.14 of [1] and Chapter 4.1 for more information about key generation functions and private key templates respectively. Additionally the attribute **CKA_AUTH_PIN_FLAGS** is supplied. The **CKA_AUTH_PIN_FLAGS** attribute will among other things be used to indicate that the PIN of the private key was verified for the next operation to be done (see Chapter 4.1).

2.1.3 Changing the secondary authentication PIN value

The application causes the device to change the secondary authentication PIN on a private key using the **C_SetAuthPIN** function. For details see Chapter 5.2.2.

3 New general data types

This chapter contains additions to Chapter 9 of [1].

3.1 New object types

This chapter contains additions to Chapter 9.4 of [1].

The following table defines the flags for secondary authentication PINs:

Bit Flag	**Mask**	**Meaning**
CKF_AUTH_PIN_COUNT_LOW	0x00000001	TRUE if an incorrect secondary authentication PIN has been entered at least once since the last successful authentication.
CKF_AUTH_PIN_FINAL_TRY	0x00000002	TRUE if supplying another incorrect secondary authentication PIN will result in the PIN becoming locked.
CKF_AUTH_PIN_LOCKED	0x00000004	TRUE if the secondary authentication PIN has been locked. The secondary PIN protected private key can no longer be used.
CKF_AUTH_PIN_TO_BE_CHANGED	0x00000008	TRUE if the secondary authentication PIN value has to be changed.
CKF_AUTH_PIN_AUTHENTICATED	0x00000010	TRUE if the verification of the secondary authentication PIN for an operation was successful. Will be reset if the operation is executed.

3.2 New data types for mechanisms

This chapter contains additions to Chapter 9.5 of [1].

The following specific mechanism types are defined[1]:

```
#define CKM_WTLS_PRE_MASTER_KEY_GEN        0x80000001
#define CKM_WTLS_MASTER_KEY_DERIVE         0x80000002
#define CKM_WTLS_MASTER_KEY_DERVIE_DH_ECC  0x80000003
#define CKM_WTLS_PRF                       0x80000004
#define CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE 0x80000005
#define CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE 0x80000006
#define CKM_TLS_PRF                        0X80000007
```

3.3 New function types

This chapter contains additions to Chapter 9.6 of [1].

[1] The numbering is to be changed. Here we use the numbers used for proprietary mechanisms.

The following Ericsson Mobile Platforms specific return values are defined:

```
#define CKR_SECONDARY_AUTHENTICATION_REQUIRED 0x80000001
```

4 Modified Objects

This chapter contains additions to Chapter 10 of [1].

4.1 Modified private key objects

This chapter contains additions to Chapter 10.9 of [1].

The following table lists the modifications to table 34 of [1]:

Attribute[*]	**Data type**	**Meaning**
CKA_SECONDARY_AUTH	CK_BBOOL	TRUE if the key requires a secondary authentication to take place before its use is allowed. (default FALSE)
CKA_AUTH_PIN_FLAGS[2,4,6]	CK_FLAGS	Mask indicating the current state of the secondary authentication PIN. If CKA_SECONDARY_AUTH is FALSE, then this attribute is zero.
CKA_AUTH_PIN[1,3,5,7,8]	UTF8 char array	Secondary authentication PIN. If CKA_SECONDARY_AUTH is FALSE, then this attribute is zero.
CKA_AUTH_PIN_LEN[1,3,5,7,8]	CK_ULONG	Length in bytes of the secondary PIN. If CKA_SECONDARY_AUTH is FALSE, then this attribute is zero.

If the **CKA_SECONDARY_AUTH** attribute is TRUE, then the Cryptoki implementation will associate the new private key object with a PIN that is stored in **CKA_AUTH_PIN** and **CKA_AUTH_PIN_LEN**. The new PIN must be presented to the token each time the key is used for a cryptographic operation. See Chapter 2.1 for the complete usage model. If **CKA_SECONDARY_AUTH** is TRUE, then **CKA_EXTRACTABLE** must be FALSE and **CKA_PRIVATE** must be TRUE. Attempts to copy private keys with **CKA_SECONDARY_AUTH** set to TRUE in a manner that would violate the above conditions must fail. An application can determine whether the setting the **CKA_SECONDARY_AUTH** attribute to TRUE is supported by checking to see if the **CKF_SECONDARY_AUTHENTICATION** flag is set in the **CK_TOKEN_INFO** flags.

The **CKA_AUTH_PIN_FLAGS** attribute indicates the current state of the secondary authentication PIN. This value is only valid if the **CKA_SECONDARY_AUTH** attribute is TRUE. The valid flags for this attribute are **CKF_AUTH_PIN_COUNT_LOW**, **CKF_AUTH_PIN_FINAL_TRY**, **CKF_AUTH_PIN_LOCKED**, **CKF_AUTH_PIN_TO_BE_CHANGED** and **CKF_AUTH_PIN_AUTHENTICATED** defined in Chapter 3.1. **CKF_AUTH_PIN_COUNT_LOW** and **CKF_AUTH_PIN_FINAL_TRY** may always be set to FALSE if the token does not support the functionality or will not reveal the information because of its security policy. The **CKF_AUTH_PIN_TO_BE_CHANGED** flag may always be FALSE if the token does not support the functionality.

[*] The meaning of the footnotes in this column is described in table 24 of [1].

The **CKA_AUTH_PIN** and **CKA_AUTH_PIN_LEN** attributes contain the secondary authentication PIN itself. They are only valid if the **CKA_SECONDARY_AUTH** attribute is TRUE.

5 New functions

This chapter contains additions to Chapter 11 of [1].

5.1 New function return values

This chapter contains additions to Chapter 11.1 of [1].

- CKR_SECONDARY_AUTHENTICATION_REQUIRED: This value can only be returned by the function **C_SignInit**. It means that the private key used for signing is protected by secondary authentication. Thus the secondary authentication PIN has to be verified before the signing operation can be executed.

5.2 New object management functions

This chapter contains additions to Chapter 11.7 of [1].

5.2.1 C_AuthenticateObject

```
CK_DEFINE_FUNCTION(CK_RV, C_AuthenticateObject)(
  CK_SESSION_HANDLE hSession,
  CK_OBJECT_HANDLE hPrivateKeyObject,
  CK_UTF8CHAR_PTR pPin,
  CK_ULONG ulPinLen
);
```

C_AuthenticateObject verifies the secondary authentication PIN of a private key object. *hSession* is the session's handle; *hPrivateKeyObject* is the private key object's handle, *pPin* points to the PIN that shall be verified against the secondary authentication PIN; *ulPinLen* is the length of the PIN to be verified. This standard allows PIN values to contain any valid UTF8 character, but the token may impose subset restrictions.

If the token has a "protected authentication path", as indicated by the **CKF_PROTECTED_AUTHENTICATION_PATH** flag in its **CK_TOKEN_INFO** being set, then that means that there is some way to be authenticated to the private key object without having the application send a PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PINpad on the token itself, or on the slot device. Or the user might not even use a PIN - authentication could be achieved by some fingerprint-reading device, for example. To log into a token with a protected authentication path, the *pPin* parameter to **C_AuthenticateObject** should be NULL_PTR. When **C_AuthenticateObject** returns, whatever authentication method supported by the token will have been performed; a return value of CKR_OK means that the user was successfully authenticated, and a return value of CKR_PIN_INCORRECT means that the user was denied access.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_PIN_EXPIRED, CKR_PIN_INCORRECT, CKR_PIN_INVALID, CKR_PIN_LOCKED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY_EXISTS.

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE  hPrivateKeyObject;
CK_UTF8CHAR       PIN[] = {"MyPIN"};
CK_RV             rv;
```

```
rv = C_AuthenticateObject(hSession, hPrivateKeyObject, PIN,
        sizeof(PIN));
if (rv == CKR_OK)
{
  .
  .
  .
  }
}
```

5.2.2 C_SetAuthPIN

```
CK_DEFINE_FUNCTION(CK_RV, C_SetAuthPIN)(
  CK_SESSION_HANDLE hSession,
  CK_OBJECT_HANDLE  hPrivateKeyObject,
  CK_UTF8CHAR_PTR   pOldPin,
  CK_ULONG          ulOldLen,
  CK_UTF8CHAR_PTR   pNewPin,
  CK_ULONG          ulNewLen
);
```

C_SetAuthPIN modifies the secondary authentication PIN of the specified private key object. *hSession* is the session's handle; *hPrivateKeyObject* is the private key object's handle, *pOldPin* points to the old PIN; *ulOldLen* is the length in bytes of the old PIN; *pNewPin* points to the new PIN; *ulNewLen* is the length in bytes of the new PIN. This standard allows PIN values to contain any valid UTF8 character, but the token may impose subset restrictions.

C_SetAuthPIN can only be called in the "R/W User Functions" state. An attempt to call it from a session in any other state fails with error CKR_SESSION_READ_ONLY or CKR_USER_NOT_LOGGED_IN depending on the sessions state.

If the token has a "protected authentication path", as indicated by the CKF_PROTECTED_AUTHENTICATION_PATH flag in its **CK_TOKEN_INFO** being set, then that means that there is some way some way to be authenticated to the private key object without having the application send a PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PINpad on the token itself, or on the slot device. To modify the specified secondary authentication PIN on a token with such a protected authentication path, the *pOldPin* and *pNewPin* parameters to **C_SetAuthPIN** should be NULL_PTR. During the execution of **C_SetAuthPIN**, the current user will enter the old PIN and the new PIN through the protected authentication path. It is not specified how the PINpad should be used to enter *two* PINs; this varies.

If the token has a protected authentication path other than a PINpad, then it is token-dependent whether or not **C_SetAuthPIN** can be used to modify the specified secondary authentication PIN.

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_PIN_INCORRECT, CKR_PIN_INVALID, CKR_PIN_LEN_RANGE, CKR_PIN_LOCKED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_USER_NOT_LOGGED_IN, CKR_TOKEN_WRITE_PROTECTED, CKR_ARGUMENTS_BAD.

Example:

```
CK_SESSION_HANDLE hSession;
```

```
CK_OBJECT_HANDLE hPrivateKeyObject;
CK_UTF8CHAR oldPin[] = {“OldPIN”};
CK_UTF8CHAR newPin[] = {“NewPIN”};
CK_RV rv;

rv = C_SetPIN(hSession, hPrivateKeyObject, oldPin,
        sizeof(oldPin), newPin, sizeof(newPin));
if (rv == CKR_OK) {
  .
  .
  .
}
```

5.3 Modified signing and MACing functions

This chapter contains additions to Chapter 11.11 of [1].

5.3.1 C_SignInit

An additional return value: CKR_SECONDARY_AUTHENTICATION_REQUIRED

6 New mechanisms

This chapter contains additions to Chapter 12 of [1].

6.1 TLS mechanism parameters

Details can be found in [3].

6.1.1 CK_TLS_PRF_PARAMS

CK_TLS_PRF_PARAMS is a structure, which provides the parameters to the **CKM_TLS_PRF** mechanism. It is defined as follows:

```
typedef struct
{
  CK_BYTE_PTR pLabel;
  CK_ULONG    ulLabelLen;
  CK_ULONG    ulOutputLen;
} CK_TLS_PRF_PARAMS;
```

The fields of the structure have the following meanings:

pLabel	pointer to the identifying label
ulLabelLen	length in bytes of the identifying label
ulOutputLen	length in bytes that the output to be created shall have

CK_TLS_PRF_PARAMS_PTR is a pointer to a **CK_TLS_PRF_PARAMS**.

6.2 TLS mechanisms

Details can be found in [3].

6.2.1 PRF (pseudo random function)

PRF (pseudo random function) in TLS, denoted **CKM_TLS_PRF**, is a mechanism used to produce a secure digest protected by a secret key. It is used to produce an output of arbitrary length. The keys it uses are generic secret keys.

It has a parameter, a **CK_TLS_PRF_PARAMS** structure, which provides an identifying label that is linked with the input seed.

6.3 WTLS mechanism parameters

Details can be found in [2].

6.3.1 CK_WTLS_RANDOM_DATA

CK_WTLS_RANDOM_DATA is a structure, which provides information about the random data of a client and a server in a WTLS context. This structure is used by the **CKM_WTLS_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```
typedef struct
{
  CK_BYTE_PTR pClientRandom;
  CK_ULONG    ulClientRandomLen;
  CK_BYTE_PTR pServerRandom;
  CK_ULONG    ulServerRandomLen;
} CK_WTLS_RANDOM_DATA;
```

The fields of the structure have the following meanings:

pClientRandom	pointer to the clients random data
ulClientRandomLen	length in bytes of the clients random data
pServerRandom	pointer to the servers random data
ulServerRandomLen	length in bytes of the servers random data

6.3.2 CK_WTLS_MASTER_KEY_DERIVE_PARAMS

CK_WTLS_MASTER_KEY_DERIVE_PARAMS is a structure, which provides the parameters to the **CKM_WTLS_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```
typedef struct
{
  CK_MECHANISM_TYPE   DigestMechansim;
  CK_WTLS_RANDOM_DATA RandomInfo;
  CK_BYTE_PTR         pVersion;
} CK_WTLS_MASTER_KEY_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

DigestMechanism	the mechanism type of the digest mechanism to be used (possible types can be found in [2])
RandomInfo	clients and servers random data information
pVersion	pointer to a **CK_BYTE** which receives the WTLS protocol version information

CK_WTLS_MASTER_KEY_DERIVE_PARAMS_PTR is a pointer to a **CK_WTLS_MASTER_KEY_DERIVE_PARAMS**.

6.3.3 CK_WTLS_PRF_PARAMS

CK_WTLS_PRF_PARAMS is a structure, which provides the parameters to the **CKM_WTLS_PRF** mechanism. It is defined as follows:

```
typedef struct
{
  CK_MECHANISM_TYPE DigestMechanism;
  CK_BYTE_PTR       pLabel;
  CK_ULONG          ulLabelLen;
  CK_ULONG          ulOutputLen;
} CK_WTLS_PRF_PARAMS;
```

The fields of the structure have the following meanings:

DigestMechanism	the mechanism type of the digest mechanism to be used (possible types can be found in [2])

pLabel	pointer to the identifying label
ulLabelLen	length in bytes of the identifying label
ulOutputLen	length in bytes that the output to be created shall have

CK_WTLS_PRF_PARAMS_PTR is a pointer to a **CK_WTLS_PRF_PARAMS**.

6.3.4 CK_WTLS_KEY_MAT_OUT

CK_WTLS_KEY_MAT_OUT is a structure that contains the resulting key handles and initialization vectors after performing a C_DeriveKey function with the **CKM_WTLS_SEVER_KEY_AND_MAC_DERIVE** or with the **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE** mechanism. It is defined as follows:

```
typedef struct
{
  CK_OBJECT_HANDLE hMacSecret;
  CK_OBJECT_HANDLE hKey;
  CK_BYTE_PTR      pIV;
} CK_WTLS_KEY_MAT_OUT;
```

The fields of the structure have the following meanings:

hMacSecret	key handle for the resulting MAC secret key
hKey	key handle for the resulting secret key
pIV	Pointer to a location which receives the initialisation vector (IV) created (if any)

CK_WTLS_KEY_MAT_OUT _PTR is a pointer to a **CK_WTLS_KEY_MAT_OUT**.

6.3.5 CK_WTLS_KEY_MAT_PARAMS

CK_WTLS_KEY_MAT_PARAMS is a structure that provides the parameters to the **CKM_WTLS_SEVER_KEY_AND_MAC_DERIVE** and the **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE** mechanisms. It is defined as follows:

```
typedef struct
{
  CK_MECHANISM_TYPE       DigestMechanism;
  CK_ULONG                ulMacSizeInBits;
  CK_ULONG                ulKeySizeInBits;
  CK_ULONG                ulIVSizeInBits;
  CK_ULONG                ulSequenceNumber;
  CK_BBOOL                bIsExport;
  CK_WTLS_RANDOM_DATA     RandomInfo;
  CK_WTLS_KEY_MAT_OUT_PTR pReturnedKeyMaterial;
} CK_WTLS_KEY_MAT_PARAMS;
```

The fields of the structure have the following meanings:

DigestMechanism	the mechanism type of the digest mechanism to be used (possible types can be found in [2])
ulMacSizeInBits	the length (in bits) of the MACing keys agreed

	upon during the protocol handshake phase
ulKeySizeInBits	the length (in bits) of the secret keys agreed upon during the handshake phase
ulIVSizeInBits	the length (in bits) of the IV agreed upon during the handshake phase. If no IV is required, the length should be set to 0.
ulSequenceNumber	The current sequence number used for records sent by the client and server respectively
bIsExport	a boolean value which indicates whether the keys have to be derived for an export version of the protocol. If this value is true (i.e. the keys are exportable) then *ulKeySizeInBits* is the length of the key in bits before expansion. The length of the key after expansion is determined by the information found in the template sent along with this mechanism during a C_DeriveKey function call (either the **CKA_KEY_TYPE** or the **CKA_VALUE_LEN** attribute).
RandomInfo	client's and server's random data information
pReturnedKeyMaterial	points to a **CK_WTLS_KEY_MAT_OUT** structure which receives the handles for the keys generated and the IVs

CK_WTLS_KEY_MAT_PARAMS_PTR is a pointer to a **CK_WTLS_KEY_MAT_PARAMS**.

6.4 WTLS mechanisms

Details can be found in [2].

6.4.1 Pre master secret key generation for RSA key exchange suite

Pre master secret key generation for the RSA key exchange suite in WTLS denoted **CKM_WTLS_PRE_MASTER_KEY_GEN**, is a mechanism, which generates a variable length secret key. It is used to produce the pre master secret key for RSA key exchange suite used in WTLS. This mechanism returns a handle to the pre master secret key.

It has one parameter, a **CK_BYTE**, which provides the client's WTLS version.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE** and **CKA_VALUE** attributes to the new key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C_GenerateKey** call may indicate that the object class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN** attribute indicates the length of the pre master secret key.

For this mechanism, the ulMinKeySize field of the **CK_MECHANISM_INFO** structure indicate 20 bytes.

6.4.2 Master secret key derivation

Master secret derivation in WTLS, denoted **CKM_WTLS_MASTER_KEY_DERIVE**, is a mechanism used to derive a 20 byte generic secret key from variable length secret key. It is used to produce the master secret key used in WTLS from the pre master secret key. This mechanism returns the value of the client version, which is built into the pre master secret key as well as a handle to the derived master secret key.

It has a parameter, a **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the mechanism type of the digest mechanism to be used as well as the passing of random data to the token as well as the returning of the protocol version number which is part of the pre master secret key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C_GenerateKey** call may indicate that the object class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN** attribute has value 20. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

This mechanism has the following rules about key sensitivity and extractability:

The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both be specified to be either TRUE or FALSE. If omitted, these attributes each take on some default value.

If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to FALSE, then the derived key will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to TRUE, then the derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its **CKA_SENSITIVE** attribute.

Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to FALSE, then the derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite* value from its **CKA_EXTRACTABLE** attribute.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure both indicate 20 bytes.

Note that the **CK_BYTE** pointed to by the **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** structure's *pVersion* field will be modified by the **C_DeriveKey** call. In particular, when the call returns, this byte will hold the WTLS version associated with the supplied pre master secret key.

Note that this mechanism is only useable for key exchange suites that use a 20-byte pre master secret key with an embedded version number. This includes the RSA key exchange suites, but excludes the Diffie-Hellman and Elliptic Curve Cryptography key exchange suites.

6.4.3 Master secret key derivation for Diffie-Hellman and Elliptic Curve Cryptography

Master secret derivation for Diffie-Hellman and Elliptic Curve Cryptography in WTLS, denoted **CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC**, is a mechanism used to derive a 20 byte generic secret key from variable length secret key. It is used to produce the master secret key used in WTLS from the pre master secret key. This mechanism returns a handle to the derived master secret key.

It has a parameter, a **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the passing of the mechanism type of the digest mechanism to be used as well as random data to the token. The *pVersion* field of the structure must be set to NULL_PTR since the version number is not embedded in the pre master secret key as it is for RSA-like key exchange suites.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C_GenerateKey** call may indicate that the object class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN** attribute has value 20. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

This mechanism has the following rules about key sensitivity and extractability:

The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both be specified to be either TRUE or FALSE. If omitted, these attributes each take on some default value.

If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to FALSE, then the derived key will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to TRUE, then the derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its **CKA_SENSITIVE** attribute.

Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to FALSE, then the derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite* value from its **CKA_EXTRACTABLE** attribute.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure both indicate 20 bytes.

Note that this mechanism is only useable for key exchange suites that do not use a fixed length 20-byte pre master secret key with an embedded version number. This includes the Diffie-Hellman and Elliptic Curve Cryptography key exchange suites, but excludes the RSA key exchange suites.

6.4.4 PRF (pseudo random function)

PRF (pseudo random function) in WTLS, denoted **CKM_WTLS_PRF**, is a mechanism used to produce a secure digest protected by a secret key. It is used to produce an output of arbitrary length. The keys it uses are generic secret keys.

It has a parameter, a **CK_WTLS_PRF_PARAMS** structure, which provides the mechanism type of the digest mechanism to be used and an identifying label that is linked with the input seed.

6.4.5 Server Key and MAC derivation

Server key, MAC and IV derivation in WTLS, denoted **CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE**, is a mechanism used to derive the appropriate cryptographic keying material used by a cipher suite from the master secret key and random data. This mechanism returns the key handles for the keys generated in the process, as well as the IV created.

It has a parameter, a **CK_WTLS_KEY_MAT_PARAMS** structure, which allows for the passing of the mechanism type of the digest mechanism to be used as well as random data as well as the characteristic of the cryptographic material for the given cipher suite and a pointer to a structure which receives the handles and IV which were generated. This structure is defined in Section 6.3.4

This mechanism contributes to the creation of two distinct keys on the token and returns one IV (if an IV is requested by the caller) back to the caller. The keys are all given an object class of **CKO_SECRET_KEY**.

The MACing key (server write MAC secret) is always given a type of **CKK_GENERIC_SECRET**. It is flagged as valid for signing, verification and derivation operations.

The other key (server write key) is typed according to information found in the template sent along with this mechanism during a **C_DeriveKey** function call. By default, it is flagged as valid for encryption, decryption, and derivation operations.

An IV (server write IV) will be generated and returned if the *ulIVSizeInBits* field of the **CK_WTLS_KEY_MAT_PARAMS** field has a nonzero value. If it is generated, its length in bits will agree with the value in the *ulIVSizeInBits* field

Both keys inherit the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**, **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The template provided to **C_DeriveKey** may not specify values for any of these attributes that differ from those held by the base key.

Note that the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the **CK_WTLS_KEY_MAT_PARAMS** structure's *pReturnedKeyMaterial* field will be modified by the **C_DeriveKey** call. In particular, the two key handle fields in the **CK_WTLS_KEY_MAT_OUT** structure will be modified to hold handles to the newly-created keys; in addition, the buffer pointed to by the **CK_WTLS_KEY_MAT_OUT** structure's *pIV* field will have the IV returned in them (if an IV is requested by the caller). Therefore, this field must point to a buffer with sufficient space to hold any IV that will be returned.

This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information. For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result of a successful completion. However, since the **CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE** mechanism returns all of its key handles in the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the **CK_WTLS_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter *phKey* passed to **C_DeriveKey** is unnecessary, and should be a NULL_PTR.

If a call to **C_DeriveKey** with this mechanism fails, then *none* of the two keys will be created on the token.

6.4.6 Client key and MAC derivation

Client key, MAC and IV derivation in WTLS, denoted **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE**, is a mechanism used to derive the appropriate cryptographic keying material used by a cipher suite from the master secret key and random data. This mechanism returns the key handles for the keys generated in the process, as well as the IV created.

For this mechanism all applies as described in the Chapter 6.4.5 except for that the names server write MAC secret, server write key and server write IV have to be replaced by client write MAC secret, client write key and client write IV.

REMARK: When comparing the existing TLS mechanisms in Cryptoki with these extensions to support WTLS one could argue that there would be no need to have distinct handling of the client and server side of the handshake. However since in WTLS the server and client have different sequence numbers for the server and the client. There could be instances where WTLS is used to protect asynchronous protocols and where sequence numbers on the client and server side therefore would not be necessarily aligned, and hence this motivates the introduced split..