Amendments Proposals to PKCS#11

for support of

WTLS and TLS PRF

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 for WTLS. We also propose an amendment for TLS support. New data types and mechanisms are described.

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
TBD	To be defined
TLS	Transport Layer Security
WIM	Wireless Identification Module
WTLS	Wireless Transport Layer Security

1.2 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/
4	PKCS #15 v1.1: Cryptographic Token Information Syntax Standard	RSA Laboratories June 6, 2000 http://www.rsasecurity.com/rsalabs/PKCS/pkcs-15/index.html
5	Java MIDP 2.0 Specification.	Java Community Process http://jcp.org/jsr/detail/118.jsp

1.3 Yet to do

- -

2 New general data types

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

2.1 New object types

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

The following additional certificate type is defined.

```
#define CKC_WTLS TBD
```

2.2 New data types for mechanisms

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

The following additional mechanism types are defined.

```
#define CKM_WTLS_PRE_MASTER_KEY_GEN        TBD
#define CKM_WTLS_MASTER_KEY_DERIVE         TBD
#define CKM_WTLS_MASTER_KEY_DERVIE_DH_ECC  TBD
#define CKM_WTLS_PRF                       TBD
#define CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE TBD
#define CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE TBD
#define CKM_TLS_PRF                        TBD
```

3 New objects

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

3.1 Modified and new certificate objects

This chapter replaces Chapter 10.6 of [1].

The following figure illustrates details of certificate objects and replaces figure 7 of [1]:

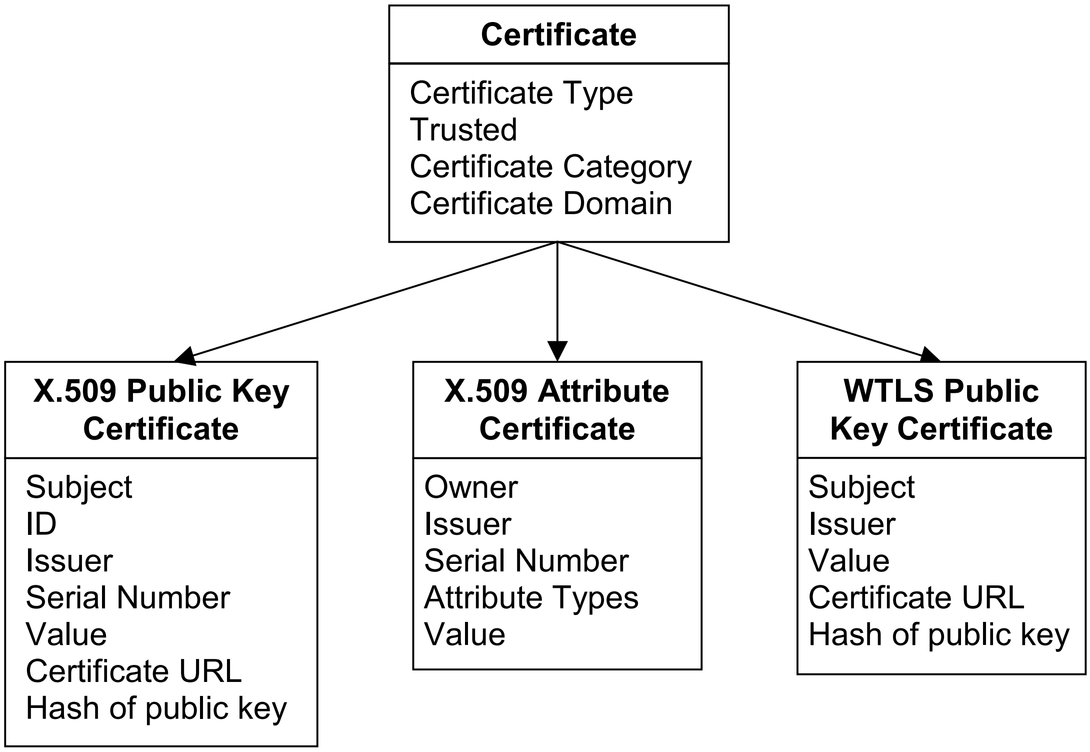

Figure 1, Certificate Object Attribute Hierarchy

Certificate objects (object class CKO_CERTIFICATE) hold public-key or attribute certificates. Other than providing access to certificate objects, Cryptoki does not attach any special meaning to certificates. The following table defines the common certificate object attributes, in addition to the common attributes listed in Table 15of [1] and Table19 of [1]:

Table: Common Certificate Object Attributes

Attribute	Data type	Meaning
CKA_CERTIFICATE_TYPE[1]	CK_CERTIFICATE_ TYPE	Type of certificate
CKA_TRUSTED	CK_BBOOL	The certificate can be trusted for the application that it was created.
CKA_CERTIFICATE_CATEGORY	CK_ULONG	The certificate is categorized under certificates as defined in [4] (0 = unspecified, 1= trusted, 2 = user, 3 = useful))
CKA_CERTIFICATE_DOMAIN	CK_ULONG	The certificate is a certificate as defined in [5] (0 = unspecified, 1 = manufacturer, 2 = operator, 3 = third_party)

[1]Must be specified when the object is created. The **CKA_CERTIFICATE_TYPE** attribute may not be modified after an object is created.

The **CKA_TRUSTED** attribute cannot be set to TRUE by an application. It must be set by a token initialization application. Trusted certificates cannot be modified.

3.1.1 X.509 public key certificate objects

This chapter replaces chapter 10.6.1 of [1] (which is not correctly numbered in [1]).

X.509 certificate objects (certificate type CKC_X_509) hold X.509 public key certificates. The following table defines the X.509 certificate object attributes, in addition to the common attributes listed in Table 15 of [1], Table19 of [1] and Table 21of [1]:

Table: X.509 Certificate Object Attributes

Attribute	Data type	Meaning
CKA_SUBJECT[1]	Byte array	DER-encoding of the certificate subject name
CKA_ID	Byte array	Key identifier for public/private key pair (default empty)
CKA_ISSUER	Byte array	DER-encoding of the certificate issuer name (default empty)
CKA_SERIAL_NUMBER	Byte array	DER-encoding of the certificate serial number (default empty)
CKA_VALUE[1]	Byte array	BER-encoding of the certificate
CKA_CERTIFICATE_URL	CK_BBOOL	TRUE if only a URL to retrieve the certificate is stored in the CKA_VALUE attribute (default FALSE)
CKA_HASH_OF_PUBLIC_KEY	Byte array	SHA-1 hash of the subjects public key as defined in [4] (default empty)

[1]Must be specified when the object is created.

Only the CKA_ID, CKA_ISSUER, and CKA_SERIAL_NUMBER attributes may be modified after the object is created.

The CKA_ID attribute is intended as a means of distinguishing multiple public-key/private-key pairs held by the same subject (whether stored in the same token or not). (Since the keys are distinguished by subject name as well as identifier, it is possible that keys for different subjects may have the same CKA_ID value without introducing any ambiguity.)

It is intended in the interests of interoperability that the subject name and key identifier for a certificate will be the same as those for the corresponding public and private keys (though it is not required that all be stored in the same token). However, Cryptoki does not enforce this association, or even the uniqueness of the key identifier for a given subject; in particular, an application may leave the key identifier empty.

The CKA_ISSUER and CKA_SERIAL_NUMBER attributes are for compatibility with PKCS #7 and Privacy Enhanced Mail (RFC1421). Note that with the version 3 extensions to X.509 certificates, the key identifier may be carried in the certificate. It is intended that the CKA_ID value be identical to the key identifier in such a certificate extension, although this will not be enforced by Cryptoki.

The CKA_CERTIFICATE_URL attribute enables the support for storage of certificate URL's.

The following is a sample template for creating a certificate object:

```
CK_OBJECT_CLASS class = CKO_CERTIFICATE;
CK_CERTIFICATE_TYPE certType = CKC_X_509;
CK_UTF8CHAR label[] = "A certificate object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE certificate[] = {...};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] = {
  {CKA_CLASS, &class, sizeof(class)},
  {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)};
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_LABEL, label, sizeof(label)-1},
```

```
  {CKA_SUBJECT, subject, sizeof(subject)},
  {CKA_ID, id, sizeof(id)},
  {CKA_VALUE, certificate, sizeof(certificate)}
};
```

3.1.2 X.509 attribute certificate objects

This chapter remains unchanged compared to chapter 10.6.2 of [1] (which is not correctly numbered in [1]).

3.1.3 WTLS public key certificate objects

Details can be found in [2].

WTLS certificate objects (certificate type **CKC_WTLS**) hold WTLS public key certificates. The following table defines the WTLS certificate object attributes, in addition to the common attributes listed in Table 15 of [1], Table19 of [1] and Table 21of [1]:

Table: WTLS Certificate Object Attributes

Attribute	Data type	Meaning
CKA_SUBJECT[1]	Byte array	WTLS-encoding of the certificate subject name.
CKA_ISSUER	Byte array	WTLS-encoding of the certificate issuer name. (default empty)
CKA_VALUE[1]	Byte array	WTLS-encoding of the certificate.
CKA_CERTIFICATE_URL	CK_BBOOL	TRUE if only a URL to retrieve the certificate is stored in the CKA_VALUE attribute (default FALSE)
CKA_HASH_OF_PUBLIC_KEY	Byte array	SHA-1 hash of the subjects public key as defined in [4] (default empty)

[1]Must be specified when the object is created.

Only the **CKA_ISSUER** attribute may be modified after the object is created.

The CKA_CERTIFICATE_URL attribute enables the support for storage of certificate URL's.

The following is a sample template for creating a certificate object:

```
CK_OBJECT_CLASS class = CKO_CERTIFICATE;
CK_CERTIFICATE_TYPE certType = CKC_WTLS;
CK_UTF8CHAR label[] = "A certificate object";
CK_BYTE subject[] = {...};
CK_BYTE certificate[] = {...};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] =
{
  {CKA_CLASS, &class, sizeof(class)},
  {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)};
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_LABEL, label, sizeof(label)-1},
  {CKA_SUBJECT, subject, sizeof(subject)},
  {CKA_VALUE, certificate, sizeof(certificate)}
};
```

4 New mechanisms

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

4.1 TLS mechanism parameters

Details can be found in [3].

4.1.1 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  pSeed;
  CK_ULONG     ulSeedLen;
  CK_BYTE_PTR  pLabel;
  CK_ULONG     ulLabelLen;
  CK_BYTE_PTR  pOutput
  CK_ULONG_PTR pulOutputLen;
} CK_TLS_PRF_PARAMS;
```

The fields of the structure have the following meanings:

pSeed	pointer to the input seed
ulSeedLen	length in bytes of the input seed
pLabel	pointer to the identifying label
ulLabelLen	length in bytes of the identifying label
pOutput	pointer receiving the output of the operation
pulOutputLen	pointer to the length in bytes that the output to be created shall have, has to hold the desired length as input and will receive the calculated length as output

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

4.2 TLS mechanisms

Details can be found in [3].

4.2.1 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 a securely generated random output of arbitrary length. The keys it uses are generic secret keys.

It has a parameter, a **CK_TLS_PRF_PARAMS** structure, which allows for the passing of the input seed and its length, the passing of an identifying label and its length and the passing of the length of the output to the token and for receiving the output.

This mechanism produces securely generated random output of the length specified in the parameter.

This mechanism departs from the other key derivation mechanisms in Cryptoki in not using the template sent along with this mechanism during a **C_DeriveKey** function call, which means the template shall be a NULL_PTR, and 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_TLS_PRF** mechanism returns the requested number of output bytes in the **CK_TLS_PRF_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 no output will be generated.

4.3 WTLS mechanism parameters

Details can be found in [2].

4.3.1 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

4.3.2 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   DigestMechanism;
  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**.

4.3.3 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       pSeed;
  CK_ULONG          ulSeedLen;
  CK_BYTE_PTR       pLabel;
  CK_ULONG          ulLabelLen;
  CK_BYTE_PTR       pOutput
  CK_ULONG_PTR      pulOutputLen;
} 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])
pSeed	pointer to the input seed
ulSeedLen	length in bytes of the input seed
pLabel	pointer to the identifying label
ulLabelLen	length in bytes of the identifying label
pOutput	pointer receiving the output of the operation
pulOutputLen	pointer to the length in bytes that the output to be created shall have, has to hold the desired length as input and will receive the calculated length as output

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

4.3.4 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**.

4.3.5 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 key agreed upon during the protocol handshake phase

ulKeySizeInBits the length (in bits) of the secret key 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 IV

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

4.4 WTLS mechanisms

Details can be found in [2].

4.4.1 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.

4.4.2 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 passing 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_DeriveKey** 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.

4.4.3 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_DeriveKey** 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.

4.4.4 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 a securely generated random output of arbitrary length. The keys it uses are generic secret keys.

It has a parameter, a **CK_WTLS_PRF_PARAMS** structure, which allows for passing the mechanism type of the digest mechanism to be used, the passing of the input seed and its length, the passing of an identifying label and its length and the passing of the length of the output to the token and for receiving the output.

This mechanism produces securely generated random output of the length specified in the parameter.

This mechanism departs from the other key derivation mechanisms in Cryptoki in not using the template sent along with this mechanism during a **C_DeriveKey** function call, which means the template shall be a NULL_PTR, and 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_PRF** mechanism returns the requested number of output bytes in the **CK_WTLS_PRF_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 no output will be generated.

4.4.5 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 4.3.4

This mechanism contributes to the creation of two distinct keys 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.

4.4.6 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 4.4.5 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.