1. C\_DeriveKeys

CK\_DEFINE\_FUNCTION(CK\_RV, C\_DeriveKeys)(
 CK\_SESSION\_HANDLE hSession,
 CK\_MECHANISM\_PTR pMechanism,
 CK\_OBJECT\_HANDLE hBaseKey,
 CK\_ATTRIBUTE\_PTR\_PTR pTemplates,
 CK\_ULONG\_PTR pulAttributeCounts,

 CK\_ULONG ulKeyCount,
 CK\_OBJECT\_HANDLE\_PTR phKeys
);

**C\_DeriveKeys** derives one or more keys from a base key, creating new key objects. *hSession* is the session’s handle; *pMechanism* points to a structure that specifies the key derivation mechanism; *hBaseKey* is the handle of the base key; *pTemplates* is an array of pointers to templates (i.e. arrays of attributes) for each of the new keys; *pulAttributeCounts* is an array of the number of attributes in each key template; *ulKeyCount* is the number of keys to be derive; and *phKeys* is an array of key handles that is used to receive the handles of the derived keys.

To process this function, enough raw key data is derived as per the description of the key derivation mechanism to meet the total requirements for all new keys and then divided up among the new keys in the order provided by the call. E.g., if 2 AES keys of length 256 bits each are required, derive 64 bytes of key data and assign the first 32 bytes to the first key and the next 32 bytes to the second key.

The values of the **CK\_SENSITIVE**, **CK\_ALWAYS\_SENSITIVE**, **CK\_EXTRACTABLE**, and **CK\_NEVER\_EXTRACTABLE** attributes for the base key affect the values that these attributes can hold for the newly-derived key. See the description of each particular key-derivation mechanism in Section for any constraints of this type.

If a call to **C\_DeriveKeys** cannot support any of the precise templates supplied to it, it will fail and return without creating any key objects.

The key objects created by a successful call to **C\_DeriveKeys** will have their **CKA\_LOCAL** attributes set to CK\_FALSE.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_ATTRIBUTE\_READ\_ONLY, CKR\_ATTRIBUTE\_TYPE\_INVALID, CKR\_ATTRIBUTE\_VALUE\_INVALID, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_DOMAIN\_PARAMS\_INVALID, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_HANDLE\_INVALID, CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SESSION\_READ\_ONLY, CKR\_TEMPLATE\_INCOMPLETE, CKR\_TEMPLATE\_INCONSISTENT, CKR\_TOKEN\_WRITE\_PROTECTED, CKR\_USER\_NOT\_LOGGED\_IN.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hPublicKey, hPrivateKey, hKey;

CK\_MECHANISM keyPairMechanism = {

 CKM\_DH\_PKCS\_KEY\_PAIR\_GEN, NULL\_PTR, 0

};

CK\_BYTE prime[] = {...};

CK\_BYTE base[] = {...};

CK\_BYTE publicValue[128];

CK\_BYTE otherPublicValue[128];

CK\_MECHANISM mechanism = {

 CKM\_DH\_PKCS\_DERIVE, otherPublicValue, sizeof(otherPublicValue)

};

CK\_ATTRIBUTE pTemplate[] = {

 CKA\_VALUE, &publicValue, sizeof(publicValue)}

};

CK\_OBJECT\_CLASS keyClass = CKO\_SECRET\_KEY;

CK\_KEY\_TYPE keyType = CKK\_DES;

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE publicKeyTemplate[] = {

 {CKA\_PRIME, prime, sizeof(prime)},

 {CKA\_BASE, base, sizeof(base)}

};

CK\_ATTRIBUTE privateKeyTemplate[] = {

 {CKA\_DERIVE, &true, sizeof(true)}

};

CK\_ATTRIBUTE template1[] = {

 {CKA\_CLASS, &keyClass, sizeof(keyClass)},

 {CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

 {CKA\_ENCRYPT, &true, sizeof(true)},

 {CKA\_DECRYPT, &true, sizeof(true)}

};

CK\_ATTRIBUTE template2[] = {

 {CKA\_CLASS, &keyClass, sizeof(keyClass)},

 {CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

 {CKA\_ENCRYPT, &true, sizeof(true)},

 {CKA\_DECRYPT, &true, sizeof(true)}

};

CK\_ATTRIBUTE\_PTR templates[] = { template1, template2 };

CK\_ULONG attributeCounts[] = { 4, 4};

CK\_OBJECT\_HANDLE newKeys[2];

CK\_RV rv;

.

.

rv = C\_GenerateKeyPair(

 hSession, &keyPairMechanism,

 publicKeyTemplate, 2,

 privateKeyTemplate, 1,

 &hPublicKey, &hPrivateKey);

if (rv == CKR\_OK) {

 rv = C\_GetAttributeValue(hSession, hPublicKey, &pTemplate, 1);

 if (rv == CKR\_OK) {

 /\* Put other guy’s public value in otherPublicValue \*/

 .

 .

 rv = C\_DeriveKey(

 hSession, &mechanism,

 hPrivateKey, templates, attributeCounts,

 2, newKeys);

 if (rv == CKR\_OK) {

 .

 .

 }

 }

}

**CK\_GENERIC\_DERIVE\_PARAMS** is a structure that provides the parameters to the **CKM\_GENERIC\_DERIVE** mechanism. It is defined as follows:

typedef struct CK\_GENERIC\_DERIVE\_PARAMS {

 CK\_MECHANISM\_TYPE mechType;

 CK\_BYTE\_PTR pPublicData;

 CK\_ULONG\_PTR pulPublicDataLengths;

 CK\_ULONG ulPublicDataCount;

} CK\_GENERIC\_DERIVE\_PARAMS;

The fields of the structure have the following meanings:

 *mechType* the mechanism to use to derive key material.

 *pPublicData* the public data used as part of the key derivation mechanism. This is the concatenation of all of the public data used to derive key material for this mechanism described by mechType.

 *pulPublicDataLengths* the lengths of each component of the public data.

 *ulPublicDataCount* the number of components of the public data.

**CK\_GENERIC\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_GENERIC\_PARAMS**.

[To be done: Expand the description to be more general. For now, here's how it plays with TLS1.1 and beyond.

CK\_GENERIC\_DERIVE\_PARAMS tlsParams;

tlsParams.mechType = CKM\_TLS\_10\_DERIVE;

tlsParams.pPublicData = malloc(length\_of\_seed + length\_of\_label);

// copy seed and label to pPublicData

CK\_ULONG[] componentLengths = { length\_of\_seed, length\_of\_label };

tlsParams.pulPublicDataLengths = componentLengths;

tlsParams.ulPublicDataCount = 2;

If the label value is "master secret" you should use the pre\_master secret as the base key and generate a single key.

If the label value is "key expansion" you should use the master secret as the base key and generate 2 encryption keys and 2 mac keys.

TLS "Final" value.

### TLS MAC

The TLS MAC mechanisms are used to generate integrity tags for the TLS "finished" message.

**CKM\_TLS10\_MAC\_SERVER**

**CKM\_TLS10\_MAC\_CLIENT**

**CML\_TLS12\_MAC**

**CKM\_TLS10\_MAC\_SERVER** and **TLS10\_MAC\_CLIENT** take no parameters. They both calculate the verification data as described for TLS1.0 and 1.1 with **CKM\_TLS10\_MAC\_SERVER** using "server finished" as the label, and **CKM\_TLS10\_MAC\_CLIENT** using "client finished" as the label.

**CKM\_TLS12\_MAC** This length should be in the range 0-20 (the output size of SHA-1 is 20 bytes). Signatures (MACs) produced by this mechanism will be taken from the start of the full 20-byte HMAC output.

Table xx, General-length TLS MAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | generic secret | any | 12 bytes |
| C\_Verify | generic secret | any | 12 bytes |

**CK\_TLS12\_MAC\_PARAMS** is a structure that provides the parameters to the **CKM\_TLS12\_MAC** mechanism. It is defined as follows:

typedef struct CK\_TLS12\_MAC\_PARAMS {

 CK\_MECHANISM\_TYPE prfHashFunction;

 CK\_ULONG ulServerOrClient;

} CK\_TLS12\_MAC\_PARAMS;

The fields of the structure have the following meanings:

 *prfHashFunction* the hash mechanism used in the TLS12 PRF construct. If the mechanism type CKM\_TLS\_PRF is used, this mechanism is should return the same data as one of the CKM\_TLS10\_\* mechanisms.

 *ulServerOrClient* 1 to use the label "server finished", 2 to use the label "client finished". All other values are invalid.

**CK\_TLS12\_MAC\_PARAMS\_PTR** is a pointer to a **CK\_TLS12\_MAC\_PARAMS**.

If accepted, **CKM\_TLS\_PRF** shall be deprecated/prohibited as a derivation mechanism. **CKM\_TLS\_MASTER\_KEY\_DERIVE** shall be deprecated, but permitted. **CKM\_TLS\_KEY\_AND\_MAC\_DERIVE** shall be deprecated, but permitted only if it does not allow the generation of "public" data (e.g. no data from the KDF shall be placed in the IV fields).