EDDSA proposal by introducing additional EC key types

draft 2, Nov 15, 2017

Author: Darren Johnson

Summary:

This is one of multiple proposals created to present possible options for including RFC 8032 (Ed25519 and Ed448) and RFC 7748 (Curve25519 and Curve 448) in PKCS #11

This proposal creates two additional key types. One key type for Edwards curves (RFC 8032) and one key type for Montgomery curves (RFC 7748), with the understanding that the existing EC framework is focussed on curves represented in Weierstrass form.

Utimaco comments and suggestions

**U1: Curve and algorithm names**

We suggest to strictly use the curve and algorithm names as defined in the RFCs:

* “curve25519”, “curve448”, “edwards25519” and “edwards448” (with lowercase initial) for curves in Montgomery / Edwards representation, as defined in RFC 7748.
* “Ed25519” and “Ed448” (with uppercase initial) for EdDSA algorithms as defined in RFC 8032.

I have updated the wording throughout the document where needed.

**U2: Curve selection in CKA\_EC\_PARAMS**

It is our understanding that none of the existing choices in CKA\_EC\_PARAMS can be used as is for Edwards/Montgomery curves:

* ecParameters does not fit for Edwards/Montgomery curves
* namedCurve specifies an Object Identifier, but there are currently no Object Identifier for Edwards/Montgomery curves
* implicilyCA must not be used

We would thus

* either need Object Identifier for “curve25519”, “curve448”, “edwards25519” and “edwards448” => add new OIDs in the OASIS tree of OIDs, or contact another organization to define such OIDs;
* or add another choice to CKA\_EC\_PARAMS, e.g. curveName being a string with the curve names as defined in RFC 7748. (Yet “curveName” can easily be confused with “namedCurve”). In this case, table 2 below (table 29 in the original standard) must be extended with a flag CKF\_EC\_CURVENAME.

When choosing to extend the CHOICE by “curveName”, all occurences of “Edwards/Montgomery EC public/private keys only support the use of the namedCurve selection” must be updated.

**U3: Encoding**

Should we explicitly mention that encoding of keys and messages must be in little endian format? Although this is specified in RFC 7748/8032, mentioning it in the PKCS#11 standard makes it more visible, as in other chapters that also explicitly mention endianness. Such explicit statement would apply to all definitions of CKA\_EC\_POINT for Edwards and Montgomery keys.

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| Revision | Changes | Author | Date |
| Draft 1 | Initial Draft | Darren Johnson | Jan 23, 2017 |
| Draft 2 | Finished missing/incomplete sections | Darren Johnson | Nov 15, 2017 |

## Elliptic Curve

The Elliptic Curve (EC) cryptosystem (also related to ECDSA) in this document was originally based on the one described in the ANSI X9.62 and X9.63 standards developed by the ANSI X9F1 working group.

The EC cryptosystem developed by the ANSI X9F1 working group was created at a time when EC curves were always represented in their Weierstrass form. Since that time, new curves represented in Edwards form (RFC 8032) and Montgomery form (RFC 7748) have become more common. To support these new curves, the EC cryptosystem in this document has been extended from the original. Additional key generation mechanisms have been added as well as an additional signature generation mechanism.

*Table 1, Elliptic Curve Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_EC\_KEY\_PAIR\_GEN (CKM\_ECDSA\_KEY\_PAIR\_GEN) |  |  |  |  | ✓ |  |  |
| CKM\_EC\_EDWARDS\_KEY\_PAIR\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_EC\_MONTGOMERY\_KEY\_PAIR\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_ECDSA |  | ✓2 |  |  |  |  |  |
| CKM\_ECDSA\_SHA1 |  | ✓ |  |  |  |  |  |
| CKM\_EDDSA |  | ✓ |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| CKM\_ECDH1\_DERIVE |  |  |  |  |  |  | ✓ |
| CKM\_ECDH1\_COFACTOR\_DERIVE |  |  |  |  |  |  | ✓ |
| CKM\_ECMQV\_DERIVE |  |  |  |  |  |  | ✓ |
| CKM\_ECDH\_AES\_KEY\_WRAP |  |  |  |  |  |  |  |

Table 2, Mechanism Information Flags

|  |  |  |
| --- | --- | --- |
| CKF\_EC\_F\_P | 0x00100000UL | True if the mechanism can be used with EC domain parameters over *Fp* |
| CKF\_EC\_F\_2M | 0x00200000UL | True if the mechanism can be used with EC domain parameters over *F*2*m* |
| CKF\_EC\_ECPARAMETERS | 0x00400000UL | True if the mechanism can be used with EC domain parameters of the choice **ecParameters** |
| CKF\_EC\_NAMEDCURVE | 0x00800000UL | True if the mechanism can be used with EC domain parameters of the choice **namedCurve** |
| CKF\_EC\_UNCOMPRESS | 0x01000000UL | True if the mechanism can be used with elliptic curve point uncompressed |
| CKF\_EC\_COMPRESS | 0x02000000UL | True if the mechanism can be used with elliptic curve point compressed |

In these standards, there are two different varieties of EC defined:

1. EC using a field with an odd prime number of elements (i.e. the finite field *Fp*).
2. EC using a field of characteristic two (i.e. the finite field *F*2*m*).

An EC key in Cryptoki contains information about which variety of EC it is suited for. It is preferable that a Cryptoki library, which can perform EC mechanisms, be capable of performing operations with the two varieties of EC, however this is not required. The **CK\_MECHANISM\_INFO** structure **CKF\_EC\_F\_P** flag identifies a Cryptoki library supporting EC keys over *Fp* whereas the **CKF\_EC\_F\_2M** flag identifies a Cryptoki library supporting EC keys over *F*2*m*. A Cryptoki library that can perform EC mechanisms must set either or both of these flags for each EC mechanism.

In these specifications there are also three representation methods to define the domain parameters for an EC key. Only the **ecParameters** and the **namedCurve** choices are supported in Cryptoki. The **CK\_MECHANISM\_INFO** structure **CKF\_EC\_ECPARAMETERS** flag identifies a Cryptoki library supporting the **ecParameters** choice whereas the **CKF\_EC\_NAMEDCURVE** flag identifies a Cryptoki library supporting the **namedCurve** choice. A Cryptoki library that can perform EC mechanisms must set either or both of these flags for each EC mechanism.

In these specifications, an EC public key (i.e. EC point *Q*) or the base point *G* when the **ecParameters** choice is used can be represented as an octet string of the uncompressed form or the compressed form. The **CK\_MECHANISM\_INFO** structure **CKF\_EC\_UNCOMPRESS** flag identifies a Cryptoki library supporting the uncompressed form whereas the **CKF\_EC\_COMPRESS** flag identifies a Cryptoki library supporting the compressed form. A Cryptoki library that can perform EC mechanisms must set either or both of these flags for each EC mechanism.

Note that an implementation of a Cryptoki library supporting EC with only one variety, one representation of domain parameters or one form may encounter difficulties achieving interoperability with other implementations.

If an attempt to create, generate, derive or unwrap an EC key of an unsupported curve is made, the attempt should fail with the error code CKR\_CURVE\_NOT\_SUPPORTED. If an attempt to create, generate, derive, or unwrap an EC key with invalid or of an unsupported representation of domain parameters is made, that attempt should fail with the error code CKR\_DOMAIN\_PARAMS\_INVALID. If an attempt to create, generate, derive, or unwrap an EC key of an unsupported form is made, that attempt should fail with the error code CKR\_TEMPLATE\_INCONSISTENT.

### EC Signatures

For the purposes of these mechanisms, an ECDSA signature is an octet string of even length which is at most two times *nLen* octets, where *nLen* is the length in octets of the base point order *n*. The signature octets correspond to the concatenation of the ECDSA values *r* and *s*, both represented as an octet string of equal length of at most *nLen* with the most significant byte first. If *r* and *s* have different octet length, the shorter of both must be padded with leading zero octets such that both have the same octet length. Loosely spoken, the first half of the signature is *r* and the second half is *s*. For signatures created by a token, the resulting signature is always of length 2*nLen*. For signatures passed to a token for verification, the signature may have a shorter length but must be composed as specified before.

If the length of the hash value is larger than the bit length of *n*, only the leftmost bits of the hash up to the length of *n* will be used. Any truncation is done by the token.

Note: For applications, it is recommended to encode the signature as an octet string of length two times *nLen* if possible. This ensures that the application works with PKCS#11 modules which have been implemented based on an older version of this document. Older versions required all signatures to have length two times *nLen*. It may be impossible to encode the signature with the maximum length of two times *nLen* if the application just gets the integer values of *r* and *s* (i.e. without leading zeros), but does not know the base point order *n*, because *r* and *s* can have any value between zero and the base point order *n*.

### Definitions

This section defines the key type “CKK\_ECDSA”, “CKK\_EC”, “CKK\_EC\_EDWARDS” and “CKK\_EC\_MONTGOMERY” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

Note: CKM\_ECDSA\_KEY\_PAIR\_GEN is deprecated in v2.11

CKM\_ECDSA\_KEY\_PAIR\_GEN

CKM\_EC\_KEY\_PAIR\_GEN

CKM\_ECDSA

CKM\_ECDSA\_SHA1

CKM\_ECDH1\_DERIVE

CKM\_ECDH1\_COFACTOR\_DERIVE

CKM\_ECMQV\_DERIVE

CKM\_ECDH\_AES\_KEY\_WRAP

CKD\_NULL

CKD\_SHA1\_KDF

### ECDSA public key objects

EC (also related to ECDSA) public key objects (object class **CKO\_PUBLIC\_KEY,** key type **CKK\_EC** or **CKK\_ECDSA**) hold EC public keys. The following table defines the EC public key object attributes, in addition to the common attributes defined for this object class:

Table 3, Elliptic Curve Public Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_EC\_PARAMS1,3 (CKA\_ECDSA\_PARAMS) | Byte array | DER-encoding of an ANSI X9.62 Parameters value |
| CKA\_EC\_POINT1,4 | Byte array | DER-encoding of ANSI X9.62 ECPoint value *Q* |

- Refer to [PKCS #11-Base] table 10 for footnotes

The **CKA\_EC\_PARAMS** or **CKA\_ECDSA\_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI X9.62 as a choice of three parameter representation methods with the following syntax:

Parameters ::= CHOICE {

ecParameters ECParameters,

namedCurve CURVES.&id({CurveNames}),

implicitlyCA NULL

}

This allows detailed specification of all required values using choice **ecParameters**, the use of a **namedCurve** as an object identifier substitute for a particular set of elliptic curve domain parameters, or **implicitlyCA** to indicate that the domain parameters are explicitly defined elsewhere. The use of a **namedCurve** is recommended over the choice **ecParameters**. The choice **implicitlyCA** must not be used in Cryptoki.

The following is a sample template for creating an EC (ECDSA) public key object:

CK\_OBJECT\_CLASS class = CKO\_PUBLIC\_KEY;

CK\_KEY\_TYPE keyType = CKK\_EC;

CK\_UTF8CHAR label[] = “An EC public key object”;

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

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

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

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

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

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

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_EC\_PARAMS, ecParams, sizeof(ecParams)},

{CKA\_EC\_POINT, ecPoint, sizeof(ecPoint)}

};

### Elliptic curve private key objects

EC (also related to ECDSA) private key objects (object class **CKO\_PRIVATE\_KEY,** key type **CKK\_EC** or **CKK\_ECDSA**) hold EC private keys. See Section 1.0 for more information about EC. The following table defines the EC private key object attributes, in addition to the common attributes defined for this object class:

Table 4, Elliptic Curve Private Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_EC\_PARAMS1,4,6 (CKA\_ECDSA\_PARAMS) | Byte array | DER-encoding of an ANSI X9.62 Parameters value |
| CKA\_VALUE1,4,6,7 | Big integer | ANSI X9.62 private value *d* |

- Refer to [PKCS #11-Base] table 10 for footnotes

The **CKA\_EC\_PARAMS** or **CKA\_ECDSA\_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI X9.62 as a choice of three parameter representation methods with the following syntax:

Parameters ::= CHOICE {

ecParameters ECParameters,

namedCurve CURVES.&id({CurveNames}),

implicitlyCA NULL

}

This allows detailed specification of all required values using choice **ecParameters**, the use of a **namedCurve** as an object identifier substitute for a particular set of elliptic curve domain parameters, or **implicitlyCA** to indicate that the domain parameters are explicitly defined elsewhere. The use of a **namedCurve** is recommended over the choice **ecParameters**. The choice **implicitlyCA** must not be used in Cryptoki.

Note that when generating an EC private key, the EC domain parameters are *not* specified in the key’s template. This is because EC private keys are only generated as part of an EC key *pair*, and the EC domain parameters for the pair are specified in the template for the EC public key.

The following is a sample template for creating an EC (ECDSA) private key object:

CK\_OBJECT\_CLASS class = CKO\_PRIVATE\_KEY;

CK\_KEY\_TYPE keyType = CKK\_EC;

CK\_UTF8CHAR label[] = “An EC private key object”;

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

CK\_BYTE id[] = {123};

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

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

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

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

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

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

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_SUBJECT, subject, sizeof(subject)},

{CKA\_ID, id, sizeof(id)},

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

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

{CKA\_EC\_PARAMS, ecParams, sizeof(ecParams)},

{CKA\_VALUE, value, sizeof(value)}

};

### Edwards Elliptic curve public key objects

Edwards EC public key objects (object class **CKO\_PUBLIC\_KEY,** key type **CKK\_EC\_EDWARDS**) hold Edwards EC public keys. The following table defines the Edwards EC public key object attributes, in addition to the common attributes defined for this object class:

Table 5, Edwards Elliptic Curve Public Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_EC\_PARAMS1,3 | Byte array | DER-encoding of an ANSI X9.62 Parameters value |
| CKA\_EC\_POINT1,4 | Byte array | DER-encoding of the b-bit public key value as defined in RFC 8032 |

- Refer to [PKCS #11-Base] table 10 for footnotes

The **CKA\_EC\_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI X9.62 as a choice of three parameter representation methods with the following syntax:

Parameters ::= CHOICE {

ecParameters ECParameters,

namedCurve CURVES.&id({CurveNames}),

implicitlyCA NULL

}

Edwards EC public keys only support the use of the **namedCurve** selection.

The following is a sample template for creating an Edwards EC public key object:

CK\_OBJECT\_CLASS class = CKO\_PUBLIC\_KEY;

CK\_KEY\_TYPE keyType = CKK\_EC;

CK\_UTF8CHAR label[] = “An Edwards EC public key object”;

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

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

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

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

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

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

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_EC\_PARAMS, ecParams, sizeof(ecParams)},

{CKA\_EC\_POINT, ecPoint, sizeof(ecPoint)}

};

### Edwards Elliptic curve private key objects

Edwards EC private key objects (object class **CKO\_PRIVATE\_KEY,** key type **CKK\_EC\_EDWARDS**) hold Edwards EC private keys. See Section 1.0 for more information about EC. The following table defines the Edwards EC private key object attributes, in addition to the common attributes defined for this object class:

Table 6, Edwards Elliptic Curve Private Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_EC\_PARAMS1,4,6 | Byte array | DER-encoding of an ANSI X9.62 Parameters value |
| CKA\_VALUE1,4,6,7 | Big integer | b-bit private key value as defined in RFC 8032 |

- Refer to [PKCS #11-Base] table 10 for footnotes

The **CKA\_EC\_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI X9.62 as a choice of three parameter representation methods with the following syntax:

Parameters ::= CHOICE {

ecParameters ECParameters,

namedCurve CURVES.&id({CurveNames}),

implicitlyCA NULL

}

Edwards EC private keys only support the use of the **namedCurve** selection.

Note that when generating an Edwards EC private key, the EC domain parameters are *not* specified in the key’s template. This is because Edwards EC private keys are only generated as part of an Edwards EC key *pair*, and the EC domain parameters for the pair are specified in the template for the Edwards EC public key.

The following is a sample template for creating an Edwards EC private key object:

CK\_OBJECT\_CLASS class = CKO\_PRIVATE\_KEY;

CK\_KEY\_TYPE keyType = CKK\_EC;

CK\_UTF8CHAR label[] = “An Edwards EC private key object”;

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

CK\_BYTE id[] = {123};

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

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

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

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

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

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

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_SUBJECT, subject, sizeof(subject)},

{CKA\_ID, id, sizeof(id)},

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

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

{CKA\_EC\_PARAMS, ecParams, sizeof(ecParams)},

{CKA\_VALUE, value, sizeof(value)}

};

### Montgomery Elliptic curve public key objects

Montgomery EC public key objects (object class **CKO\_PUBLIC\_KEY,** key type **CKK\_EC\_MONTGOMERY**) hold Montgomery EC public keys. The following table defines the Montgomery EC public key object attributes, in addition to the common attributes defined for this object class:

Table 7, Montgomery Elliptic Curve Public Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_EC\_PARAMS1,3 | Byte array | DER-encoding of an ANSI X9.62 Parameters value |
| CKA\_EC\_POINT1,4 | Byte array | DER-encoding of the public key value as defined in RFC 7748 |

- Refer to [PKCS #11-Base] table 10 for footnotes

The **CKA\_EC\_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI X9.62 as a choice of three parameter representation methods with the following syntax:

Parameters ::= CHOICE {

ecParameters ECParameters,

namedCurve CURVES.&id({CurveNames}),

implicitlyCA NULL

}

Montgomery EC public keys only support the use of the **namedCurve** selection.

The following is a sample template for creating a Montgomery EC public key object:

CK\_OBJECT\_CLASS class = CKO\_PUBLIC\_KEY;

CK\_KEY\_TYPE keyType = CKK\_EC;

CK\_UTF8CHAR label[] = “A Montgomery EC public key object”;

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

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

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

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

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

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

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_EC\_PARAMS, ecParams, sizeof(ecParams)},

{CKA\_EC\_POINT, ecPoint, sizeof(ecPoint)}

};

### Montgomery Elliptic curve private key objects

Montgomery EC private key objects (object class **CKO\_PRIVATE\_KEY,** key type **CKK\_EC\_MONTGOMERY**) hold Montgomery EC private keys. See Section 1.0 for more information about EC. The following table defines the Montgomery EC private key object attributes, in addition to the common attributes defined for this object class:

Table 8, Montgomery Elliptic Curve Private Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_EC\_PARAMS1,4,6 | Byte array | DER-encoding of an ANSI X9.62 Parameters value |
| CKA\_VALUE1,4,6,7 | Big integer | Private key value as defined in RFC 7748 |

- Refer to [PKCS #11-Base] table 10 for footnotes

The **CKA\_EC\_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI X9.62 as a choice of three parameter representation methods with the following syntax:

Parameters ::= CHOICE {

ecParameters ECParameters,

namedCurve CURVES.&id({CurveNames}),

implicitlyCA NULL

}

Edwards EC private keys only support the use of the **namedCurve** selection.

Note that when generating a Montgomery EC private key, the EC domain parameters are *not* specified in the key’s template. This is because Montgomery EC private keys are only generated as part of a Montgomery EC key *pair*, and the EC domain parameters for the pair are specified in the template for the Montgomery EC public key.

The following is a sample template for creating a Montgomery EC private key object:

CK\_OBJECT\_CLASS class = CKO\_PRIVATE\_KEY;

CK\_KEY\_TYPE keyType = CKK\_EC;

CK\_UTF8CHAR label[] = “A Montgomery EC private key object”;

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

CK\_BYTE id[] = {123};

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

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

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

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

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

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

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_SUBJECT, subject, sizeof(subject)},

{CKA\_ID, id, sizeof(id)},

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

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

{CKA\_EC\_PARAMS, ecParams, sizeof(ecParams)},

{CKA\_VALUE, value, sizeof(value)}

};

### Elliptic curve key pair generation

The EC (also related to ECDSA) key pair generation mechanism, denoted **CKM\_EC\_KEY\_PAIR\_GEN** or **CKM\_ECDSA\_KEY\_PAIR\_GEN**, is a key pair generation mechanism for EC.

This mechanism does not have a parameter.

The mechanism generates EC public/private key pairs with particular EC domain parameters, as specified in the **CKA\_EC\_PARAMS** or **CKA\_ECDSA\_PARAMS** attribute of the template for the public key. Note that this version of Cryptoki does not include a mechanism for generating these EC domain parameters.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_EC\_POINT** attributes to the new public key and the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_EC\_PARAMS** or **CKA\_ECDSA\_PARAMS** and **CKA\_VALUE** attributes to the new private key. Other attributes supported by the EC public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For example, if a Cryptoki library supports only ECDSA using a field of characteristic 2 which has between 2200 and 2300 elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation, the number 2200 consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly, 2300 is a 301-bit number).

### Edwards Elliptic curve key pair generation

The Edwards EC key pair generation mechanism, denoted **CKM\_EC\_EDWARDS\_KEY\_PAIR\_GEN**, is a key pair generation mechanism for EC keys over curves represented in Edwards form.

This mechanism does not have a parameter.

The mechanism can only generate EC public/private key pairs over the curves edwards25519 and edwards448 as defined in RFC 8032. These curves can only be specified in the **CKA\_EC\_PARAMS** attribute of the template for the public key using the **namedCurve** method. Attempts to generate keys over these curves using any other EC key pair generation mechanism will fail with CKR\_CURVE\_NOT\_SUPPORTED.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_EC\_POINT** attributes to the new public key and the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_EC\_PARAMS** and **CKA\_VALUE** attributes to the new private key. Other attributes supported by the Edwards EC public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For this mechanism, the only allowed values are 255 and 448 as RFC 8032 only defines curves of these two sizes. A Cryptoki implementation may support one or both of these curves and should set the *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

### Montgomery Elliptic curve key pair generation

The Montgomery EC key pair generation mechanism, denoted **CKM\_EC\_MONTGOMERY\_KEY\_PAIR\_GEN**, is a key pair generation mechanism for EC keys over curves represented in Montgomery form.

This mechanism does not have a parameter.

The mechanism can only generate Montgomery EC public/private key pairs over the curves curve25519 and curve448 as defined in RFC 7748. These curves can only be specified in the **CKA\_EC\_PARAMS** attribute of the template for the public key using the **namedCurve** method. Attempts to generate keys over these curves using any other EC key pair generation mechanism will fail with CKR\_CURVE\_NOT\_SUPPORTED.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_EC\_POINT** attributes to the new public key and the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_EC\_PARAMS** and **CKA\_VALUE** attributes to the new private key. Other attributes supported by the EC public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For this mechanism, the only allowed values are 255 and 448 as RFC 7748 only defines curves of these two sizes. A Cryptoki implementation may support one or both of these curves and should set the *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

### ECDSA without hashing

Refer section 1.0.1 for signature encoding.

The ECDSA without hashing mechanism, denoted **CKM\_ECDSA**, is a mechanism for single-part signatures and verification for ECDSA. (This mechanism corresponds only to the part of ECDSA that processes the hash value, which should not be longer than 1024 bits; it does not compute the hash value.)

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 9, ECDSA: Key and Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign1 | ECDSA private key | any3 | 2*nLen* |
| C\_Verify1 | ECDSA public key | any3, ≤2*nLen* 2 | N/A |

1 Single-part operations only.

2 Data length, signature length.

3 Input the entire raw digest. Internally, this will be truncated to the appropriate number of bits.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For example, if a Cryptoki library supports only ECDSA using a field of characteristic 2 which has between 2200 and 2300 elements (inclusive), then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation, the number 2200 consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly, 2300 is a 301-bit number).

### ECDSA with SHA-1

Refer to section 1.0.1 for signature encoding.

The ECDSA with SHA-1 mechanism, denoted **CKM\_ECDSA\_SHA1**, is a mechanism for single- and multiple-part signatures and verification for ECDSA. This mechanism computes the entire ECDSA specification, including the hashing with SHA-1.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 10, ECDSA with SHA-1: Key and Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign | ECDSA private key | any | 2*nLen* |
| C\_Verify | ECDSA public key | any, ≤2*nLen* 2 | N/A |

2 Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For example, if a Cryptoki library supports only ECDSA using a field of characteristic 2 which has between 2200 and 2300 elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation, the number 2200 consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly, 2300 is a 301-bit number).

### EdDSA

The EdDSA mechanism, denoted **CKM\_EDDSA**, is a mechanism for single-part and multiplart signatures and verification for EdDSA. This mechanism implements the five EdDSA signature schemes defined in RFC 8032.

This mechanism has an optional parameter, a **CK\_EDDSA\_PARAMS** structure. The absence or presence of the parameter as well as its content is used to identify which signature scheme is to be used. Table 32 enumerates the five signature schemes defined in RFC 8032 and all supported permutations of the mechanism parameter and its content.

Table 11, Mapping to RFC 8032 Signature Schemes

| **Signature Scheme** | **Mechanism Param** | **phFlag** | **Context Data** |
| --- | --- | --- | --- |
| Ed25519 | Not Required | N/A | N/A |
| Ed25519ctx | Required | False | Optional |
| Ed25519ph | Required | True | Optional |
| Ed448 | Required | False | Optional |
| Ed448ph | Required | True | Optional |

Constraints on key types and the length of data are summarized in the following table:

Table 12, EdDSA: Key and Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign | CKK\_EC\_EDWARDS private key | any | 2b*Len* |
| C\_Verify | CKK\_EC\_EDWARDS public key | any, ≤2b*Len* 2 | N/A |

2 Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For this mechanism, the only allowed values are 255 and 448 as RFC 7748 only defines curves of these two sizes. A Cryptoki implementation may support one or both of these curves and should set the *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

### EC mechanism parameters

**CK\_EDDSA\_PARAMS** is a structure that provides the parameters for the **CKM\_EDDSA** signature mechanism. The structure is defined as follows:

typedef struct CK\_EDDSA\_PARAMS {

CK\_BBOOL phFlag;

CK\_ULONG ulContextDataLen;

CK\_BYTE\_PTR pContextData;

} CK\_EDDSA\_PARAMS;

The fields of the structure have the following meanings:

phFlag a Boolean value which indicates if Prehashed variant of EdDSA should used

ulContextDataLen the length in bytes of the context data where 0 <= ulContextDataLen <= 255.

pContextData context data shared between the signer and verifier

**CK\_EDDSA\_PARAMS\_PTR** is a pointer to a **CK\_EDDSA\_PARAMS**.

* **CK\_EC\_KDF\_TYPE, CK\_EC\_KDF\_TYPE\_PTR**

**CK\_EC\_KDF\_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive keying data from a shared secret. The key derivation function will be used by the EC key agreement schemes. It is defined as follows:

typedef CK\_ULONG CK\_EC\_KDF\_TYPE;

The following table lists the defined functions.

Table 13, EC: Key Derivation Functions

|  |
| --- |
| **Source Identifier** |
| CKD\_NULL |
| CKD\_SHA1\_KDF |
| CKD\_SHA224\_KDF |
| CKD\_SHA256\_KDF |
| CKD\_SHA384\_KDF |
| CKD\_SHA512\_KDF |

The key derivation function **CKD\_NULL** produces a raw shared secret value without applying any key derivation function whereas the key derivation function **CKD\_SHA1\_KDF**, which isbased on SHA-1, derives keying data from the shared secret value as defined in ANSI X9.63.

**CK\_EC\_KDF\_TYPE\_PTR** is a pointer to a **CK\_EC\_KDF\_TYPE**.

* **CK\_ECDH1\_DERIVE\_PARAMS, CK\_ECDH1\_DERIVE\_PARAMS\_PTR**

**CK\_ECDH1\_DERIVE\_PARAMS** is a structure that provides the parameters for the **CKM\_ECDH1\_DERIVE** and **CKM\_ECDH1\_COFACTOR\_DERIVE** key derivation mechanisms, where each party contributes one key pair. The structure is defined as follows:

typedef struct CK\_ECDH1\_DERIVE\_PARAMS {

CK\_EC\_KDF\_TYPE kdf;

CK\_ULONG ulSharedDataLen;

CK\_BYTE\_PTR pSharedData;

CK\_ULONG ulPublicDataLen;

CK\_BYTE\_PTR pPublicData;

} CK\_ECDH1\_DERIVE\_PARAMS;

The fields of the structure have the following meanings:

kdf key derivation function used on the shared secret value

ulSharedDataLen the length in bytes of the shared info

pSharedData some data shared between the two parties

ulPublicDataLen the length in bytes of the other party’s EC public key

pPublicData[[1]](#footnote-2) pointer to other party’s EC public key value. A token MUST be able to accept this value encoded as a raw octet string (as per section A.5.2 of [ANSI X9.62]). A token MAY, in addition, support accepting this value as a DER-encoded ECPoint (as per section E.6 of [ANSI X9.62]) i.e. the same as a CKA\_EC\_POINT encoding. The calling application is responsible for converting the offered public key to the compressed or uncompressed forms of these encodings if the token does not support the offered form.

With the key derivation function **CKD\_NULL**, *pSharedData* must be NULL and *ulSharedDataLen* must be zero. With the key derivation function **CKD\_SHA1\_KDF**, an optional *pSharedData* may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, *pSharedData* must be NULL and *ulSharedDataLen* must be zero.

**CK\_ECDH1\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_ECDH1\_DERIVE\_PARAMS**.

* **CK\_ECMQV\_DERIVE\_PARAMS, CK\_ECMQV\_DERIVE\_PARAMS\_PTR**

**CK\_ECMQV\_DERIVE\_PARAMS** is a structure that provides the parameters to the **CKM\_ECMQV\_DERIVE** key derivation mechanism, where each party contributes two key pairs. The structure is defined as follows:

typedef struct CK\_ECMQV\_DERIVE\_PARAMS {

CK\_EC\_KDF\_TYPE kdf;

CK\_ULONG ulSharedDataLen;

CK\_BYTE\_PTR pSharedData;

CK\_ULONG ulPublicDataLen;

CK\_BYTE\_PTR pPublicData;

CK\_ULONG ulPrivateDataLen;

CK\_OBJECT\_HANDLE hPrivateData;

CK\_ULONG ulPublicDataLen2;

CK\_BYTE\_PTR pPublicData2;

CK\_OBJECT\_HANDLE publicKey;

} CK\_ECMQV\_DERIVE\_PARAMS;

The fields of the structure have the following meanings:

kdf key derivation function used on the shared secret value

ulSharedDataLen the length in bytes of the shared info

pSharedData some data shared between the two parties

ulPublicDataLen the length in bytes of the other party’s first EC public key

pPublicData pointer to other party’s first EC public key value. Encoding rules are as per pPublicData of CK\_ECDH1\_DERIVE\_PARAMS

ulPrivateDataLen the length in bytes of the second EC private key

hPrivateData key handle for second EC private key value

ulPublicDataLen2 the length in bytes of the other party’s second EC public key

pPublicData2 pointer to other party’s second EC public key value. Encoding rules are as per pPublicData of CK\_ECDH1\_DERIVE\_PARAMS

publicKey Handle to the first party’s ephemeral public key

With the key derivation function **CKD\_NULL**, *pSharedData* must be NULL and *ulSharedDataLen* must be zero. With the key derivation function **CKD\_SHA1\_KDF**, an optional *pSharedData* may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, *pSharedData* must be NULL and *ulSharedDataLen* must be zero.

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

### Elliptic curve Diffie-Hellman key derivation

The elliptic curve Diffie-Hellman (ECDH) key derivation mechanism, denoted **CKM\_ECDH1\_DERIVE**, is a mechanism for key derivation based on the Diffie-Hellman version of the elliptic curve key agreement scheme, as defined in ANSI X9.63, where each party contributes one key pair all using the same EC domain parameters.

It has a parameter, a **CK\_ECDH1\_DERIVE\_PARAMS** structure.

This mechanism derives a secret value, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template.

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 CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
* If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_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 CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_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 specify the minimum and maximum supported number of bits in the field sizes, respectively. For example, if a Cryptoki library supports only EC using a field of characteristic 2 which has between 2200 and 2300 elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation, the number 2200 consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly, 2300 is a 301-bit number).

Constraints on key types are summarized in the following table:

Table 14, ECDH: Allowed Key Types

| **Function** | **Key type** |
| --- | --- |
| C\_Derive | CKK\_EC or CKK\_EC\_MONTGOMERY |

### Elliptic curve Diffie-Hellman with cofactor key derivation

The elliptic curve Diffie-Hellman (ECDH) with cofactor key derivation mechanism, denoted **CKM\_ECDH1\_COFACTOR\_DERIVE**, is a mechanism for key derivation based on the cofactor Diffie-Hellman version of the elliptic curve key agreement scheme, as defined in ANSI X9.63, where each party contributes one key pair all using the same EC domain parameters. Cofactor multiplication is computationally efficient and helps to prevent security problems like small group attacks.

It has a parameter, a **CK\_ECDH1\_DERIVE\_PARAMS** structure.

This mechanism derives a secret value, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template.

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 CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
* If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_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 CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_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 specify the minimum and maximum supported number of bits in the field sizes, respectively. For example, if a Cryptoki library supports only EC using a field of characteristic 2 which has between 2200 and 2300 elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation, the number 2200 consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly, 2300 is a 301-bit number).

Constraints on key types are summarized in the following table:

Table 15, ECDH with Cofactor: Allowed Key Types

| **Function** | **Key type** |
| --- | --- |
| C\_Derive | CKK\_EC or CKK\_EC\_MONTGOMERY |

### Elliptic curve Menezes-Qu-Vanstone key derivation

The elliptic curve Menezes-Qu-Vanstone (ECMQV) key derivation mechanism, denoted **CKM\_ECMQV\_DERIVE**, is a mechanism for key derivation based the MQV version of the elliptic curve key agreement scheme, as defined in ANSI X9.63, where each party contributes two key pairs all using the same EC domain parameters.

It has a parameter, a **CK\_ECMQV\_DERIVE\_PARAMS** structure.

This mechanism derives a secret value, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template.

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 CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
* If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_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 CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_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 specify the minimum and maximum supported number of bits in the field sizes, respectively. For example, if a Cryptoki library supports only EC using a field of characteristic 2 which has between 2200 and 2300 elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation, the number 2200 consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly, 2300 is a 301-bit number).

Constraints on key types are summarized in the following table:

Table 16, ECDH MQV: Allowed Key Types

| **Function** | **Key type** |
| --- | --- |
| C\_Derive | CKK\_EC |

### ECDH AES KEY WRAP

The ECDH AES KEY WRAP mechanism, denoted **CKM\_ECDH\_AES\_KEY\_WRAP**, is a mechanism based on elliptic curve public-key crypto-system and the AES key wrap mechanism.  It supports single-part key wrapping; and key unwrapping.

It has a parameter, a **CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS** structure.

The mechanism can wrap and un-wrap an asymmetric target key of any length and type using an EC key.

* A temporary AES key is derived from a temporary EC key and the wrapping EC key using the **CKM\_ECDH1\_DERIVE** mechanism.
* The derived AES key is used for wrapping the target key using the **CKM\_AES\_KEY\_WRAP\_PAD** mechanism.

For wrapping, the mechanism -

* Generates a temporary random EC key (transport key) having the same parameters as the wrapping EC key (and domain parameters). Saves the transport key public key material.
* Performs ECDH operation using **CKM\_ECDH1\_DERIVE** with parameters of kdf, ulSharedDataLen and pSharedData using the private key of the transport EC key and the public key of wrapping EC key and gets the first ulAESKeyBits bits of the derived key to be the temporary AES key
* Wraps the target key with the temporary AES key using **CKM\_AES\_KEY\_WRAP\_PAD (**RFC5649).
* Zeroizes the temporary AES key and EC transport private key
* Concatenates public key material of the transport key and output the concatenated blob.

The recommended format for an asymmetric target key being wrapped is as a PKCS8 PrivateKeyInfo

The use of Attributes in the PrivateKeyInfo structure is OPTIONAL. In case of conflicts between the object attribute template, and Attributes in the PrivateKeyInfo structure, an error should be thrown.

For unwrapping, the mechanism -

* Splits the input into two parts. The first part is the public key material of the transport key and the second part is the wrapped target key. The length of the first part is equal to the length of the public key material of the unwrapping EC key

*Note: since the transport key and the wrapping EC key share the same domain, the length of the public key material of the transport key is the same length of the public key material of the unwrapping EC key.*

* Performs ECDH operation using **CKM\_ECDH1\_DERIVE** with parameters of kdf, ulSharedDataLen and pSharedData using the private part of unwrapping EC key and the public part of the transport EC key and gets first ulAESKeyBits bits of the derived key to be the temporary AES key
* Un-wraps the target key from the second part with the temporary AES key using **CKM\_AES\_KEY\_WRAP\_PAD** **(**RFC5649).
* Zeroizes the temporary AES key

*Table 17, CKM\_ECDH\_AES\_KEY\_WRAP Mechanisms vs. Functions*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Functions** | | | | | | |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_ECDH\_AES\_KEY\_WRAP |  |  |  |  |  | ✓ |  |
| 1SR = SignRecover, VR = VerifyRecover | | | | | | | |

Constraints on key types are summarized in the following table:

Table 18, ECDH AES Key Wrap: Allowed Key Types

| **Function** | **Key type** |
| --- | --- |
| C\_Derive | CKK\_EC or CKK\_EC\_MONTGOMERY |

### ECDH AES KEY WRAP mechanism parameters

1. CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS; CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS\_PTR

**CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS** is a structure that provides the parameters to the **CKM\_ECDH\_AES\_KEY\_WRAP** mechanism.  It is defined as follows:

typedef struct CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS {

CK\_ULONG                         ulAESKeyBits;

CK\_EC\_KDF\_TYPE kdf;

CK\_ULONG ulSharedDataLen;

CK\_BYTE\_PTR pSharedData;

} CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS;

The fields of the structure have the following meanings:

*ulAESKeyBits* length of the temporary AES key in bits. Can be only 128, 192 or 256.

*Kdf* key derivation function used on the shared secret value to generate AES key.

*ulSharedDataLen* the length in bytes of the shared info

*pSharedData* Some data shared between the two parties

**CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS\_PTR** is a pointer to a **CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS**.

### FIPS 186-4

When using elliptic curve mechanisms for ECDSA and ECDH in a FIPS mode of operation, any curve may be used. The only requirement NIST has is that at least one NIST recommended curve is supported for the purpose of Cryptographic Algorithm Validation Program (CAVP) testing.

CKM\_EDDSA may not be used in a FIPS mode of operation. NIST has not yet approved the use of the Edwards-Curve Digital Signature (EdDSA) signature algorithm.

1. The encoding in V2.20 was not specified and resulted in different implementations choosing different encodings. Applications relying only on a V2.20 encoding (e.g. the DER variant) other than the one specified now (raw) may not work with all V2.30 compliant tokens. [↑](#footnote-ref-2)