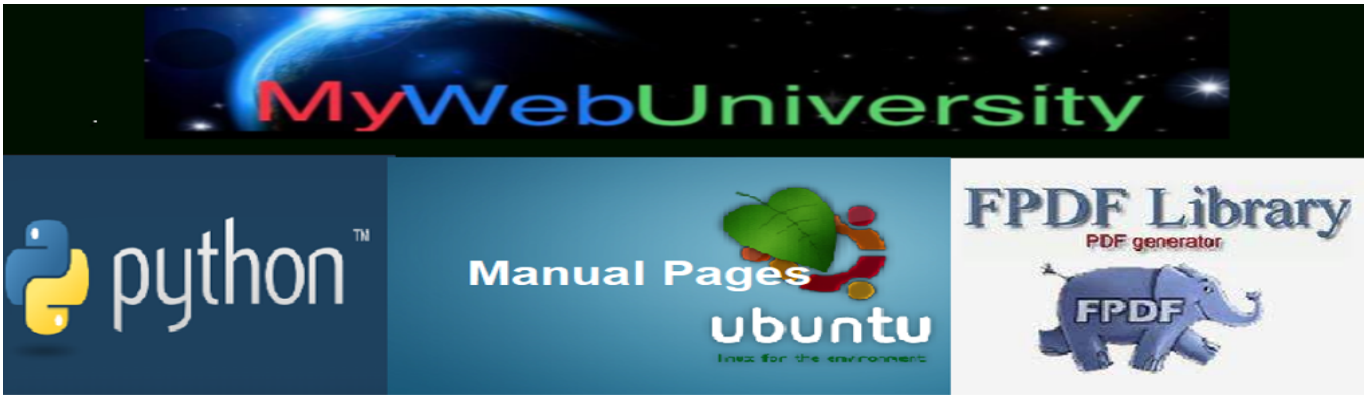

Rocky Enterprise Linux 9.2 Manual Pages on command 'Crypt::PK::ECC.3pm'

$ man Crypt::PK::ECC.3pm

Crypt::PK::ECC(3pm) User Contributed Perl Documentation Crypt::PK::ECC(3pm)

NAME

Crypt::PK::ECC - Public key cryptography based on EC

SYNOPSIS

```
### OO interface
#Encryption: Alice
my $pub = Crypt::PK::ECC->new('Bob_pub_ecc1.der');
my $ct = $pub->encrypt("secret message");
#
#Encryption: Bob (received ciphertext $ct)
my $priv = Crypt::PK::ECC->new('Bob_priv_ecc1.der');
my $pt = $priv->decrypt($ct);
#Signature: Alice
my $priv = Crypt::PK::ECC->new('Alice_priv_ecc1.der');
my $sig = $priv->sign_message($message);
#
#Signature: Bob (received $message + $sig)
my $pub = Crypt::PK::ECC->new('Alice_pub_ecc1.der');
$pub->verify_message($sig, $message) or die "ERROR";
#Shared secret
my $priv = Crypt::PK::ECC->new('Alice_priv_ecc1.der');
my $pub = Crypt::PK::ECC->new('Bob_pub_ecc1.der');
my $shared_secret = $priv->shared_secret($pub);
```

```
#Key generation
my $pk = Crypt::PK::ECC->new();
$pk->generate_key('secp160r1');
my $private_der = $pk->export_key_der('private');
my $public_der = $pk->export_key_der('public');
my $private_pem = $pk->export_key_pem('private');
my $public_pem = $pk->export_key_pem('public');
my $public_raw = $pk->export_key_raw('public');
### Functional interface
#Encryption: Alice
my $ct = ecc_encrypt('Bob_pub_ecc1.der', "secret message");
#Encryption: Bob (received ciphertext $ct)
my $pt = ecc_decrypt('Bob_priv_ecc1.der', $ct);
#Signature: Alice
my $sig = ecc_sign_message('Alice_priv_ecc1.der', $message);
#Signature: Bob (received $message + $sig)
ecc_verify_message('Alice_pub_ecc1.der', $sig, $message) or die "ERROR";
#Shared secret
my $shared_secret = ecc_shared_secret('Alice_priv_ecc1.der', 'Bob_pub_ecc1.der');
```

DESCRIPTION

The module provides a set of core ECC functions as well as implementation of ECDSA and ECDH.

Supports elliptic curves "y^2 = x^3 + a*x + b" over prime fields "Fp = Z/pZ" (binary fields not supported).

METHODS

new

```
my $pk = Crypt::PK::ECC->new();
#or
my $pk = Crypt::PK::ECC->new($priv_or_pub_key_filename);
#or
my $pk = Crypt::PK::ECC->new(\$buffer_containing_priv_or_pub_key);
```

Support for password protected PEM keys

```
my $pk = Crypt::PK::ECC->new($priv_pem_key_filename, $password);
```

```
 #or
 my $pk = Crypt::PK::ECC->new(\$buffer_containing_priv_pem_key, $password);
generate_key
 Uses Yarrow-based cryptographically strong random number generator seeded with random data
 taken from "/dev/random" (UNIX) or "CryptGenRandom" (Win32).
 $pk->generate_key($curve_name);
 #or
 $pk->generate_key($hashref_with_curve_params);
 The following predefined $curve_name values are supported:
 # curves from http://www.ecc-brainpool.org/download/Domain-parameters.pdf
 'brainpoolp160r1'
 'brainpoolp192r1'
 'brainpoolp224r1'
 'brainpoolp256r1'
 'brainpoolp320r1'
 'brainpoolp384r1'
 'brainpoolp512r1'
 # curves from http://www.secg.org/collateral/sec2_final.pdf
 'secp112r1'
 'secp112r2'
 'secp128r1'
 'secp128r2'
 'secp160k1'
 'secp160r1'
 'secp160r2'
 'secp192k1'
 'secp192r1'   ... same as nistp192, prime192v1
 'secp224k1'
 'secp224r1'   ... same as nistp224
 'secp256k1'   ... used by Bitcoin
 'secp256r1'   ... same as nistp256, prime256v1
 'secp384r1'   ... same as nistp384
 'secp521r1'   ... same as nistp521
```

```
    #curves from http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf
    'nistp192'    ... same as secp192r1, prime192v1
    'nistp224'    ... same as secp224r1
    'nistp256'    ... same as secp256r1, prime256v1
    'nistp384'    ... same as secp384r1
    'nistp521'    ... same as secp521r1
    # curves from ANS X9.62
    'prime192v1'   ... same as nistp192, secp192r1
    'prime192v2'
    'prime192v3'
    'prime239v1'
    'prime239v2'
    'prime239v3'
    'prime256v1'   ... same as nistp256, secp256r1
  Using custom curve parameters:
    $pk->generate_key({ prime    => 'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFF',
                        A        => 'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFC',
                        B        => '22123DC2395A05CAA7423DAECCC94760A7D462256BD56916',
                        Gx       => '7D29778100C65A1DA1783716588DCE2B8B4AEE8E228F1896',
                        Gy       => '38A90F22637337334B49DCB66A6DC8F9978ACA7648A943B0',
                        order    => 'FFFFFFFFFFFFFFFFFFFFFFF7A62D031C83F4294F640EC13',
                        cofactor => 1 });
  See <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>,
  <http://www.secg.org/collateral/sec2_final.pdf>,
  <http://www.ecc-brainpool.org/download/Domain-parameters.pdf>
import_key
  Loads private or public key in DER or PEM format.
    $pk->import_key($filename);
    #or
    $pk->import_key(\$buffer_containing_key);
  Support for password protected PEM keys:
    $pk->import_key($filename, $password);
    #or
```

```
$pk->import_key(\$buffer_containing_key, $password);
```

Loading private or public keys form perl hash:

```
$pk->import_key($hashref);
# the $hashref is either a key exported via key2hash
$pk->import_key({
    curve_A        => "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7FFFFFFC",
    curve_B        => "1C97BEFC54BD7A8B65ACF89F81D4D4ADC565FA45",
    curve_bits     => 160,
    curve_bytes    => 20,
    curve_cofactor => 1,
    curve_Gx       => "4A96B5688EF573284664698968C38BB913CBFC82",
    curve_Gy       => "23A628553168947D59DCC912042351377AC5FB32",
    curve_order    => "0100000000000000000001F4C8F927AED3CA752257",
    curve_prime    => "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7FFFFFFF",
    k              => "B0EE84A749FE95DF997E33B8F333E12101E824C3",
    pub_x          => "5AE1ACE3ED0AEA9707CE5C0BCE014F6A2F15023A",
    pub_y          => "895D57E992D0A15F88D6680B27B701F615FCDC0F",
});
# or with the curve defined just by name
$pk->import_key({
    curve_name => "secp160r1",
    k          => "B0EE84A749FE95DF997E33B8F333E12101E824C3",
    pub_x      => "5AE1ACE3ED0AEA9707CE5C0BCE014F6A2F15023A",
    pub_y      => "895D57E992D0A15F88D6680B27B701F615FCDC0F",
});
# or a hash with items corresponding to JWK (JSON Web Key)
$pk->import_key({
     kty => "EC",
     crv => "P-256",
     x   => "MKBCTNIcKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4",
     y   => "4Etl6SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM",
     d   => "870MB6gfuTJ4HtUnUvYMyJpr5eUZNP4Bk43bVdj3eAE",
});
```

Supported key formats:

```
 # all formats can be loaded from a file
 my $pk = Crypt::PK::ECC->new($filename);
 # or from a buffer containing the key
 my $pk = Crypt::PK::ECC->new(\$buffer_with_key);
```

? EC private keys with with all curve parameters

```
-----BEGIN EC PRIVATE KEY-----
MIIB+gIBAQQwCKEAcA6cIt6CGfyLKm57LyXWv2PgTjydrHSbvhDJTOl+7bzUW8DS
rgSdtSPONPq1oIIBWzCCAVcCAQEwPAYHKoZIzj0BAQIxAP//////////////////
/////////////////////+/////wAAAAAAAAAA////zB7BDD//////////////
/////////////////////////////v////8AAAAAAAAAAP///wEMLMxL6fiPufk
mI4Fa+P4LRkYHZxu/oFBEgMUCI9QE4daxlY5jYou0Z0qhcjt0+wq7wMVAKM1kmqj
GaJ6HQCJamdzpIJ6zaxzBGEEqofKIr6LBTeOscce8yCtdG4dO2KLp5uYWfdB4IJU
KjhVAvJdv1UpbDpUXjhydgq3NhfeSpYmLG9dnpi/kpLcKfj0Hb0omhR86doxE7Xw
uMAKYLHOHX6BnXpDHXyQ6g5fAjEA/////////////////////////////x2NN
gfQ3Ld9YGg2ySLCneuzsGWrMxSlzAgEBoWQDYgAEeGyHPLmHcszPQ9MIIYnznpzi
QbvuJtYSjCqtIGxDfzgcLcc3nCc5tBxo+qX6OJEzcWdDAC0bwplY+9Z9jHR3ylNy
ovlHoK4ItdWkVO8NH89SLSRyVuOF8N5t3CHIo93B
-----END EC PRIVATE KEY-----
```

? EC private keys with curve defined by OID (short form)

```
-----BEGIN EC PRIVATE KEY-----
MHcCAQEEIBG1c3z52T8XwMsahGVdOZWgKCQJfv+l7djuJjgetdbDoAoGCCqGSM49
AwEHoUQDQgAEoBUyo8CQAFPeYPvv78ylh5MwFZjTCLQeb042TjiMJxG+9DLFmRSM
lBQ9T/RsLLc+PmpB1+7yPAR+oR5gZn3kJQ==
-----END EC PRIVATE KEY-----
```

? EC private keys with curve defined by OID + compressed form (supported since: CryptX-0.059)

```
-----BEGIN EC PRIVATE KEY-----
MFcCAQEEIBG1c3z52T8XwMsahGVdOZWgKCQJfv+l7djuJjgetdbDoAoGCCqGSM49
AwEHoSQDIgADoBUyo8CQAFPeYPvv78ylh5MwFZjTCLQeb042TjiMJxE=
-----END EC PRIVATE KEY-----
```

? EC private keys in password protected PEM format

```
-----BEGIN EC PRIVATE KEY-----
```

Proc-Type: 4,ENCRYPTED

DEK-Info: AES-128-CBC,98245C830C9282F7937E13D1D5BA11EC

0Y85oZ2+BKXYwrkBjsZdj6gnhOAfS5yDVmEsxFCDug+R3+Kw3QvyIfO4MVo9iWoA

D7wtoRfbt2OlBaLVl553+6QrUoa2DyKf8kLHQs1x1/J7tJOMM4SCXjlrOaToQ0dT

o7fOnjQjHne16pjgBVqGilY/I79Ab85AnE4uw7vgEucBEiU0d3nrhwuS2Opnhzyx

009q9VLDPwY2+q7tXjTqnk9mCmQgsiaDJqY09wlauSukYPgVuOJFmi1VdkRSDKYZ

rUUsQvz6Q6Q+QirSlfHna+NhUgQ2eyhGszwcP6NU8iqIxI+NCwfFVuAzw539yYwS

8SICczoC/YRlaclayXuomQ==

-----END EC PRIVATE KEY-----

? EC public keys with all curve parameters

-----BEGIN PUBLIC KEY-----

MIH1MIGuBgcqhkjOPQIBMIGiAgEBMCwGByqGSM49AQECIQD/////////////////

//////////////////+///8LzAGBAEABAEHBEEEeb5mfvncu6xVoGKVzocLBwKb

/NstzijZWfKBWxb4F5hIOtp3JqPEZV2k+/wOEQio/Re0SKaFVBmcR9CP+xDUuAIh

AP////////////////////66rtzmr0igO7/SXozQNkFBAgEBA0IABITjF/nKK3jg

pjmBRXKWAv7ekR1Ko/Nb5FFPHXjH0sDrpS7qRxFALwJHv7ylGnekgfKU3vzcewNs

lvjpBYt0Yg4=

-----END PUBLIC KEY-----

? EC public keys with curve defined by OID (short form)

-----BEGIN PUBLIC KEY-----

MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEoBUyo8CQAFPeYPvv78ylh5MwFZjT

CLQeb042TjiMJxG+9DLFmRSMlBQ9T/RsLLc+PmpB1+7yPAR+oR5gZn3kJQ==

-----END PUBLIC KEY-----

? EC public keys with curve defined by OID + public point in compressed form (supported since: CryptX-0.059)

-----BEGIN PUBLIC KEY-----

MDkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDIgADoBUyo8CQAFPeYPvv78ylh5MwFZjT

CLQeb042TjiMJxE=

-----END PUBLIC KEY-----

? PKCS#8 private keys with all curve parameters

-----BEGIN PRIVATE KEY-----

MIIBMAIBADCB0wYHKoZIzj0CATCBxwIBATAkBgcqhkjOPQEBAhkA//////////

////////v//////////MEsEGP////////////////////7//////////AQYIhI9

wjlaBcqnQj2uzMlHYKfUYiVr1WkWAxUAxGloRDXes3jEtlypWR4qV2MFmi4EMQR9

KXeBAMZaHaF4NxZYjc4ri0rujiKPGJY4qQ8iY3M3M0tJ3LZqbcj5l4rKdkipQ7AC

GQD///////////////96YtAxyD9ClPZA7BMCAQEEVTBTAgEBBBiKolTGIsTgOCtl

6dpdos0LvuaExCDFyT6hNAMyAAREwaCX0VY1LZxLW3G75tmft4p9uhc0J7/+NGaP

DN3Tr7SXkT9+co2a+8KPJhQy10k=

-----END PRIVATE KEY-----

? PKCS#8 private keys with curve defined by OID (short form)

-----BEGIN PRIVATE KEY-----

MG8CAQAwEwYHKoZIzj0CAQYIKoZIzj0DAQMEVTBTAgEBBBjFP/caeQV4WO3fnWWS

f917PGzwtypd/t+hNAMyAATSg6pBT7RO6l/p+aKcrFsGuthUdfwJWS5V3NGcVt1b

lEHQYjWya2YnHaPq/iMFa7A=

-----END PRIVATE KEY-----

? PKCS#8 encrypted private keys - password protected keys (supported since: CryptX-0.059)

-----BEGIN ENCRYPTED PRIVATE KEY-----

MIGYMBwGCiqGSIb3DQEMAQMwDgQINApjTa6oFl0CAggABHi+59l4d4e6KtG9yci2

BSC65LEsQSnrnFAExfKptNU1zMFsDLCRvDeDQDbxc6HlfoxyqFL4SmH1g3RvC/Vv

NfckdL5O2L8MRnM+ljkFtV2Te4fszWcJFdd7KiNOkPpn+7sWLfzQdvhHChLKUzmz

4INKZyMv/G7VpZ0=

-----END ENCRYPTED PRIVATE KEY-----

? EC public key from X509 certificate

-----BEGIN CERTIFICATE-----

MIIBdDCCARqgAwIBAgIJAL2BBClDEnnOMAoGCCqGSM49BAMEMBcxFTATBgNVBAMM

DFRlc3QgQ2VydCBFQzAgFw0xNzEyMzAyMDMzNDFaGA8zMDE3MDUwMjIwMzM0MVow

FzEVMBMGA1UEAwwMVGVzdCBDZXJ0IEVDMFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAE

KvkL2r5xZp7RzxLQJK+6tn/7lic+L70e1fmNbHOdxRaRvbK5G0AQWrdsbjJb92Ni

lCQk2+w/i+VuS2Q3MSR5TaNQME4wHQYDVR0OBBYEFGbJkDyKgaMcIGHS8/WuqIVw

+R8sMB8GA1UdIwQYMBaAFGbJkDyKgaMcIGHS8/WuqIVw+R8sMAwGA1UdEwQFMAMB

Af8wCgYIKoZIzj0EAwQDSAAwRQIhAJtOsmrM+gJpImoynAyqTN+7myL71uxd+YeC

6ze4MnzWAiBQi5/BqEr/SQ1+BC2TPtswvJPRFh2ZvT/6Km3gKoNVXQ==

-----END CERTIFICATE-----

? SSH public EC keys

ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbmlzdHAyNT...T3xYfJIs=

? SSH public EC keys (RFC-4716 format)

---- BEGIN SSH2 PUBLIC KEY ----

Comment: "521-bit ECDSA, converted from OpenSSH"

AAAAE2VjZHNhLXNoYTItbmlzdHA1MjEAAAAIbmlzdHA1MjEAAACFBAFk35srteP9twCwYK

vU9ovMBi77Dd6lEBPrFaMEb0CZdZ5MC3nSqflGHRWkSbUpjdPdO7cYQNpK9YXHbNSO5hbU

1gFZgyiGFxwJYYz8NAjedBXMgyH4JWplK5FQm5P5cvaglltC9qkKioUXhCc67YMYBtivXl

Ue0PgIq6kbHTqbX6+5Nw==

---- END SSH2 PUBLIC KEY ----

? EC private keys in JSON Web Key (JWK) format

See <http://tools.ietf.org/html/draft-ietf-jose-json-web-key>

{

"kty":"EC",

"crv":"P-256",

"x":"MKBCTNIcKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4",

"y":"4Etl6SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM",

"d":"870MB6gfuTJ4HtUnUvYMyJpr5eUZNP4Bk43bVdj3eAE",

}

BEWARE: For JWK support you need to have JSON module installed.

? EC public keys in JSON Web Key (JWK) format

{

"kty":"EC",

"crv":"P-256",

"x":"MKBCTNIcKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4",

"y":"4Etl6SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM",

}

BEWARE: For JWK support you need to have JSON module installed.

import_key_raw

Import raw public/private key - can load data exported by "export_key_raw".

```
$pk->import_key_raw($key, $curve);
# $key .... data exported by export_key_raw()
# $curve .. curve name or hashref with curve parameters - same as by generate_key()
```

export_key_der

```
my $private_der = $pk->export_key_der('private');
```

```
    #or
    my $public_der = $pk->export_key_der('public');
   Since CryptX-0.36 "export_key_der" can also export keys in a format that does not
   explicitly contain curve parameters but only curve OID.
    my $private_der = $pk->export_key_der('private_short');
    #or
    my $public_der = $pk->export_key_der('public_short');
   Since CryptX-0.59 "export_key_der" can also export keys in "compressed" format that
   defines curve by OID + stores public point in compressed form.
    my $private_pem = $pk->export_key_der('private_compressed');
    #or
    my $public_pem = $pk->export_key_der('public_compressed');
export_key_pem
    my $private_pem = $pk->export_key_pem('private');
    #or
    my $public_pem = $pk->export_key_pem('public');
   Since CryptX-0.36 "export_key_pem" can also export keys in a format that does not
   explicitly contain curve parameters but only curve OID.
    my $private_pem = $pk->export_key_pem('private_short');
    #or
    my $public_pem = $pk->export_key_pem('public_short');
   Since CryptX-0.59 "export_key_pem" can also export keys in "compressed" format that
   defines curve by OID + stores public point in compressed form.
    my $private_pem = $pk->export_key_pem('private_compressed');
    #or
    my $public_pem = $pk->export_key_pem('public_compressed');
   Support for password protected PEM keys
    my $private_pem = $pk->export_key_pem('private', $password);
    #or
    my $private_pem = $pk->export_key_pem('private', $password, $cipher);
    # supported ciphers: 'DES-CBC'
    #                    'DES-EDE3-CBC'
    #                    'SEED-CBC'
```

```
    #                 'CAMELLIA-128-CBC'
    #                 'CAMELLIA-192-CBC'
    #                 'CAMELLIA-256-CBC'
    #                 'AES-128-CBC'
    #                 'AES-192-CBC'
    #                 'AES-256-CBC' (DEFAULT)
export_key_jwk
    Since: CryptX-0.022
    Exports public/private keys as a JSON Web Key (JWK).
     my $private_json_text = $pk->export_key_jwk('private');
     #or
     my $public_json_text = $pk->export_key_jwk('public');
    Also exports public/private keys as a perl HASH with JWK structure.
     my $jwk_hash = $pk->export_key_jwk('private', 1);
     #or
     my $jwk_hash = $pk->export_key_jwk('public', 1);
    BEWARE: For JWK support you need to have JSON module installed.
export_key_jwk_thumbprint
    Since: CryptX-0.031
    Exports the key's JSON Web Key Thumbprint as a string.
    If you don't know what this is, see RFC 7638 <https://tools.ietf.org/html/rfc7638>.
     my $thumbprint = $pk->export_key_jwk_thumbprint('SHA256');
export_key_raw
    Export raw public/private key. Public key is exported in ASN X9.62 format (compressed or
    uncompressed), private key is exported as raw bytes (padded with leading zeros to have the
    same size as the ECC curve).
     my $pubkey_octets  = $pk->export_key_raw('public');
     #or
     my $pubckey_octets = $pk->export_key_raw('public_compressed');
     #or
     my $privkey_octets = $pk->export_key_raw('private');
encrypt
     my $pk = Crypt::PK::ECC->new($pub_key_filename);
```

```
    my $ct = $pk->encrypt($message);
    #or
    my $ct = $pk->encrypt($message, $hash_name);
    #NOTE: $hash_name can be 'SHA1' (DEFAULT), 'SHA256' or any other hash supported by Crypt::Digest
decrypt
    my $pk = Crypt::PK::ECC->new($priv_key_filename);
    my $pt = $pk->decrypt($ciphertext);
sign_message
    my $pk = Crypt::PK::ECC->new($priv_key_filename);
    my $signature = $priv->sign_message($message);
    #or
    my $signature = $priv->sign_message($message, $hash_name);
    #NOTE: $hash_name can be 'SHA1' (DEFAULT), 'SHA256' or any other hash supported by Crypt::Digest
sign_message_rfc7518
   Since: CryptX-0.024
   Same as sign_message only the signature format is as defined by
   <https://tools.ietf.org/html/rfc7518> (JWA - JSON Web Algorithms).
   BEWARE: This creates signatures according to the structure that RFC 7518 describes but
   does not apply the RFC logic for the hashing algorithm selection. You'll still need to
   specify, e.g., SHA256 for a P-256 key to get a fully RFC-7518-compliant signature.
verify_message
    my $pk = Crypt::PK::ECC->new($pub_key_filename);
    my $valid = $pub->verify_message($signature, $message)
    #or
    my $valid = $pub->verify_message($signature, $message, $hash_name);
    #NOTE: $hash_name can be 'SHA1' (DEFAULT), 'SHA256' or any other hash supported by Crypt::Digest
verify_message_rfc7518
   Since: CryptX-0.024
   Same as verify_message only the signature format is as defined by
   <https://tools.ietf.org/html/rfc7518> (JWA - JSON Web Algorithms).
   BEWARE: This verifies signatures according to the structure that RFC 7518 describes but
   does not apply the RFC logic for the hashing algorithm selection. You'll still need to
   specify, e.g., SHA256 for a P-256 key to get a fully RFC-7518-compliant signature.
```

sign_hash

```
my $pk = Crypt::PK::ECC->new($priv_key_filename);
my $signature = $priv->sign_hash($message_hash);
```

sign_hash_rfc7518

Since: CryptX-0.059

Same as sign_hash only the signature format is as defined by <https://tools.ietf.org/html/rfc7518> (JWA - JSON Web Algorithms).

verify_hash

```
my $pk = Crypt::PK::ECC->new($pub_key_filename);
my $valid = $pub->verify_hash($signature, $message_hash);
```

verify_hash_rfc7518

Since: CryptX-0.059

Same as verify_hash only the signature format is as defined by <https://tools.ietf.org/html/rfc7518> (JWA - JSON Web Algorithms).

shared_secret

```
# Alice having her priv key $pk and Bob's public key $pkb
my $pk  = Crypt::PK::ECC->new($priv_key_filename);
my $pkb = Crypt::PK::ECC->new($pub_key_filename);
my $shared_secret = $pk->shared_secret($pkb);
# Bob having his priv key $pk and Alice's public key $pka
my $pk = Crypt::PK::ECC->new($priv_key_filename);
my $pka = Crypt::PK::ECC->new($pub_key_filename);
my $shared_secret = $pk->shared_secret($pka);  # same value as computed by Alice
```

is_private

```
my $rv = $pk->is_private;
# 1 .. private key loaded
# 0 .. public key loaded
# undef .. no key loaded
```

size

```
my $size = $pk->size;
# returns key size in bytes or undef if no key loaded
```

key2hash

```
my $hash = $pk->key2hash;
```

```
    # returns hash like this (or undef if no key loaded):
    {
      size          => 20, # integer: key (curve) size in bytes
      type          => 1,  # integer: 1 .. private, 0 .. public
      #curve parameters
      curve_A       => "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7FFFFFFC",
      curve_B       => "1C97BEFC54BD7A8B65ACF89F81D4D4ADC565FA45",
      curve_bits    => 160,
      curve_bytes   => 20,
      curve_cofactor => 1,
      curve_Gx      => "4A96B5688EF573284664698968C38BB913CBFC82",
      curve_Gy      => "23A628553168947D59DCC912042351377AC5FB32",
      curve_name    => "secp160r1",
      curve_order   => "0100000000000000000001F4C8F927AED3CA752257",
      curve_prime   => "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7FFFFFF",
      #private key
      k             => "B0EE84A749FE95DF997E33B8F333E12101E824C3",
      #public key point coordinates
      pub_x         => "5AE1ACE3ED0AEA9707CE5C0BCE014F6A2F15023A",
      pub_y         => "895D57E992D0A15F88D6680B27B701F615FCDC0F",
    }
curve2hash
   Since: CryptX-0.024
    my $crv = $pk->curve2hash;
    # returns a hash that can be passed to: $pk->generate_key($crv)
    {
      A       => "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7FFFFFC",
      B       => "1C97BEFC54BD7A8B65ACF89F81D4D4ADC565FA45",
      cofactor => 1,
      Gx      => "4A96B5688EF573284664698968C38BB913CBFC82",
      Gy      => "23A628553168947D59DCC912042351377AC5FB32",
      order   => "0100000000000000000001F4C8F927AED3CA752257",
      prime   => "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7FFFFFF",
```

}

FUNCTIONS

ecc_encrypt

Elliptic Curve Diffie-Hellman (ECDH) encryption as implemented by libtomcrypt. See method "encrypt" below.

```
my $ct = ecc_encrypt($pub_key_filename, $message);
#or
my $ct = ecc_encrypt(\$buffer_containing_pub_key, $message);
#or
my $ct = ecc_encrypt($pub_key_filename, $message, $hash_name);
#NOTE: $hash_name can be 'SHA1' (DEFAULT), 'SHA256' or any other hash supported by Crypt::Digest
```

ECCDH Encryption is performed by producing a random key, hashing it, and XOR'ing the digest against the plaintext.

ecc_decrypt

Elliptic Curve Diffie-Hellman (ECDH) decryption as implemented by libtomcrypt. See method "decrypt" below.

```
my $pt = ecc_decrypt($priv_key_filename, $ciphertext);
#or
my $pt = ecc_decrypt(\$buffer_containing_priv_key, $ciphertext);
```

ecc_sign_message

Elliptic Curve Digital Signature Algorithm (ECDSA) - signature generation. See method "sign_message" below.

```
my $sig = ecc_sign_message($priv_key_filename, $message);
#or
my $sig = ecc_sign_message(\$buffer_containing_priv_key, $message);
#or
my $sig = ecc_sign_message($priv_key, $message, $hash_name);
```

ecc_verify_message

Elliptic Curve Digital Signature Algorithm (ECDSA) - signature verification. See method "verify_message" below.

```
ecc_verify_message($pub_key_filename, $signature, $message) or die "ERROR";
#or
ecc_verify_message(\$buffer_containing_pub_key, $signature, $message) or die "ERROR";
```

#or

ecc_verify_message($pub_key, $signature, $message, $hash_name) or die "ERROR";

ecc_sign_hash

Elliptic Curve Digital Signature Algorithm (ECDSA) - signature generation. See method "sign_hash" below.

my $sig = ecc_sign_hash($priv_key_filename, $message_hash);

#or

my $sig = ecc_sign_hash(\$buffer_containing_priv_key, $message_hash);

ecc_verify_hash

Elliptic Curve Digital Signature Algorithm (ECDSA) - signature verification. See method "verify_hash" below.

ecc_verify_hash($pub_key_filename, $signature, $message_hash) or die "ERROR";

#or

ecc_verify_hash(\$buffer_containing_pub_key, $signature, $message_hash) or die "ERROR";

ecc_shared_secret

Elliptic curve Diffie-Hellman (ECDH) - construct a Diffie-Hellman shared secret with a private and public ECC key. See method "shared_secret" below.

#on Alice side

my $shared_secret = ecc_shared_secret('Alice_priv_ecc1.der', 'Bob_pub_ecc1.der');

#on Bob side

my $shared_secret = ecc_shared_secret('Bob_priv_ecc1.der', 'Alice_pub_ecc1.der');

OpenSSL interoperability

let's have:

ECC private key in PEM format - eckey.priv.pem

ECC public key in PEM format - eckey.pub.pem

data file to be signed - input.data

Sign by OpenSSL, verify by Crypt::PK::ECC

Create signature (from commandline):

openssl dgst -sha1 -sign eckey.priv.pem -out input.sha1-ec.sig input.data

Verify signature (Perl code):

use Crypt::PK::ECC;

use Crypt::Digest 'digest_file';

use Crypt::Misc 'read_rawfile';

```
my $pkec = Crypt::PK::ECC->new("eckey.pub.pem");
my $signature = read_rawfile("input.sha1-ec.sig");
my $valid = $pkec->verify_hash($signature, digest_file("SHA1", "input.data"), "SHA1", "v1.5");
print $valid ? "SUCCESS" : "FAILURE";
```

Sign by Crypt::PK::ECC, verify by OpenSSL

Create signature (Perl code):

```
use Crypt::PK::ECC;
use Crypt::Digest 'digest_file';
use Crypt::Misc 'write_rawfile';
my $pkec = Crypt::PK::ECC->new("eckey.priv.pem");
my $signature = $pkec->sign_hash(digest_file("SHA1", "input.data"), "SHA1", "v1.5");
write_rawfile("input.sha1-ec.sig", $signature);
```

Verify signature (from commandline):

```
openssl dgst -sha1 -verify eckey.pub.pem -signature input.sha1-ec.sig input.data
```

Keys generated by Crypt::PK::ECC

Generate keys (Perl code):

```
use Crypt::PK::ECC;
use Crypt::Misc 'write_rawfile';
my $pkec = Crypt::PK::ECC->new;
$pkec->generate_key('secp160k1');
write_rawfile("eckey.pub.der",  $pkec->export_key_der('public'));
write_rawfile("eckey.priv.der", $pkec->export_key_der('private'));
write_rawfile("eckey.pub.pem",  $pkec->export_key_pem('public'));
write_rawfile("eckey.priv.pem", $pkec->export_key_pem('private'));
write_rawfile("eckey-passwd.priv.pem", $pkec->export_key_pem('private', 'secret'));
```

Use keys by OpenSSL:

```
openssl ec -in eckey.priv.der -text -inform der
openssl ec -in eckey.priv.pem -text
openssl ec -in eckey-passwd.priv.pem -text -inform pem -passin pass:secret
openssl ec -in eckey.pub.der -pubin -text -inform der
openssl ec -in eckey.pub.pem -pubin -text
```

Keys generated by OpenSSL

Generate keys:

```
openssl ecparam -param_enc explicit -name prime192v3 -genkey -out eckey.priv.pem
openssl ec -param_enc explicit -in eckey.priv.pem -out eckey.pub.pem -pubout
openssl ec -param_enc explicit -in eckey.priv.pem -out eckey.priv.der -outform der
openssl ec -param_enc explicit -in eckey.priv.pem -out eckey.pub.der -outform der -pubout
openssl ec -param_enc explicit -in eckey.priv.pem -out eckey.privc.der -outform der -conv_form compressed
openssl ec -param_enc explicit -in eckey.priv.pem -out eckey.pubc.der -outform der -pubout -conv_form compressed
openssl ec -param_enc explicit -in eckey.priv.pem -passout pass:secret -des3 -out eckey-passwd.priv.pem
```

Load keys (Perl code):

```
use Crypt::PK::ECC;
my $pkec = Crypt::PK::ECC->new;
$pkec->import_key("eckey.pub.der");
$pkec->import_key("eckey.pubc.der");
$pkec->import_key("eckey.priv.der");
$pkec->import_key("eckey.privc.der");
$pkec->import_key("eckey.pub.pem");
$pkec->import_key("eckey.priv.pem");
$pkec->import_key("eckey-passwd.priv.pem", "secret");
```

SEE ALSO

- ? <https://en.wikipedia.org/wiki/Elliptic_curve_cryptography>
- ? <https://en.wikipedia.org/wiki/Elliptic_curve_Diffie%E2%80%93Hellman>
- ? <https://en.wikipedia.org/wiki/ECDSA>