12.17.4 Enterprise Encryption Function Descriptions

Enterprise Encryption functions have these general characteristics:

- For arguments of the wrong type or an incorrect number of arguments, each function returns an error.
- If the arguments are not suitable to permit a function to perform the requested operation, it returns `NULL` or 0 as appropriate. This occurs, for example, if a function does not support a specified algorithm, a key length is too short or long, or a string expected to be a key string in PEM format is not a valid key.
- The underlying SSL library takes care of randomness initialization.

Several of the functions take an encryption algorithm argument. The following table summarizes the supported algorithms by function.

Table 12.22 Supported Algorithms by Function

Function	Supported Algorithms
`ASYMMETRIC_DECRYPT()`	RSA
`ASYMMETRIC_DERIVE()`	DH
`ASYMMETRIC_ENCRYPT()`	RSA
`ASYMMETRIC_SIGN()`	RSA, DSA
`ASYMMETRIC_VERIFY()`	RSA, DSA
`CREATE_ASYMMETRIC_PRIV_KEY()`	RSA, DSA, DH
`CREATE_ASYMMETRIC_PUB_KEY()`	RSA, DSA, DH

> **Note**
> Although you can create keys using any of the RSA, DSA, or DH encryption algorithms, other functions that take key arguments might accept only certain types of keys. For example, `ASYMMETRIC_ENCRYPT()` and `ASYMMETRIC_DECRYPT()` accept only RSA keys.

The following descriptions describe the calling sequences for Enterprise Encryption functions. For additional examples and discussion, see Section 12.17.2, “Enterprise Encryption Usage and Examples”.

- `ASYMMETRIC_DECRYPT(algorithm, crypt_str, key_str)`

  Decrypts an encrypted string using the given algorithm and key string, and returns the resulting cleartext as a binary string. If decryption fails, the result is `NULL`.

  `key_str` must be a valid key string in PEM format. For successful decryption, it must be the public or private key string corresponding to the private or public key string used with `ASYMMETRIC_ENCRYPT()`to produce the encrypted string. ***`algorithm`*** indicates the encryption algorithm used to create the key.

  Supported ***`algorithm`*** values: `'RSA'`

  For a usage example, see the description of `ASYMMETRIC_ENCRYPT()`.

- `ASYMMETRIC_DERIVE(`*`pub_key_str`*`, `*`priv_key_str`*`)`

  Derives a symmetric key using the private key of one party and the public key of another, and returns the resulting key as a binary string. If key derivation fails, the result is `NULL`.

  pub_key_str and ***priv_key_str*** must be valid key strings in PEM format. They must be created using the DH algorithm.

  Suppose that you have two pairs of public and private keys:

  ```
  SET @dhp = CREATE_DH_PARAMETERS(1024);
  SET @priv1 = CREATE_ASYMMETRIC_PRIV_KEY('DH', @dhp);
  SET @pub1 = CREATE_ASYMMETRIC_PUB_KEY('DH', @priv1);
  SET @priv2 = CREATE_ASYMMETRIC_PRIV_KEY('DH', @dhp);
  SET @pub2 = CREATE_ASYMMETRIC_PUB_KEY('DH', @priv2);
  ```

  Suppose further that you use the private key from one pair and the public key from the other pair to create a symmetric key string. Then this symmetric key identity relationship holds:

  ```
  ASYMMETRIC_DERIVE(@pub1, @priv2) = ASYMMETRIC_DERIVE(@pub2, @priv1)
  ```

- `ASYMMETRIC_ENCRYPT(`*`algorithm`*`, `*`str`*`, `*`key_str`*`)`

  Encrypts a string using the given algorithm and key string, and returns the resulting ciphertext as a binary string. If encryption fails, the result is `NULL`.

  The ***str*** length cannot be greater than the ***key_str*** length − 11, in bytes

  key_str must be a valid key string in PEM format. ***algorithm*** indicates the encryption algorithm used to create the key.

  Supported ***algorithm*** values: `'RSA'`

  To encrypt a string, pass a private or public key string to `ASYMMETRIC_ENCRYPT()`. To recover the original unencrypted string, pass the encrypted string to `ASYMMETRIC_DECRYPT()`, along with the public or private key string correponding to the private or public key string used for encryption.

  ```
  -- Generate private/public key pair
  SET @priv = CREATE_ASYMMETRIC_PRIV_KEY('RSA', 1024);
  SET @pub = CREATE_ASYMMETRIC_PUB_KEY('RSA', @priv);

  -- Encrypt using private key, decrypt using public key
  SET @ciphertext = ASYMMETRIC_ENCRYPT('RSA', 'The quick brown fox', @priv);
  SET @cleartext = ASYMMETRIC_DECRYPT('RSA', @ciphertext, @pub);

  -- Encrypt using public key, decrypt using private key
  SET @ciphertext = ASYMMETRIC_ENCRYPT('RSA', 'The quick brown fox', @pub);
  ```

```
SET @cleartext = ASYMMETRIC_DECRYPT('RSA', @ciphertext, @priv);
```

Suppose that:

```
SET @s = a string to be encrypted
SET @priv = a valid private RSA key string in PEM format
SET @pub = the corresponding public RSA key string in PEM format
```

Then these identity relationships hold:

```
ASYMMETRIC_DECRYPT('RSA', ASYMMETRIC_ENCRYPT('RSA', @s, @priv), @pub) = @s
ASYMMETRIC_DECRYPT('RSA', ASYMMETRIC_ENCRYPT('RSA', @s, @pub), @priv) = @s
```

- **`ASYMMETRIC_SIGN(`*`algorithm, digest_str, priv_key_str, digest_type`*`)`**

  Signs a digest string using a private key string, and returns the signature as a binary string. If signing fails, the result is **`NULL`**.

  `digest_str` is the digest string. It can be generated by calling **`CREATE_DIGEST()`**. ***`digest_type`***indicates the digest algorithm used to generate the digest string.

  `priv_key_str` is the private key string to use for signing the digest string. It must be a valid key string in PEM format. ***`algorithm`*** indicates the encryption algorithm used to create the key.

  Supported ***`algorithm`*** values: **`'RSA'`**, **`'DSA'`**

  Supported ***`digest_type`*** values: **`'SHA224'`**, **`'SHA256'`**, **`'SHA384'`**, **`'SHA512'`**

  For a usage example, see the description of **`ASYMMETRIC_VERIFY()`**.

- **`ASYMMETRIC_VERIFY(`*`algorithm, digest_str, sig_str, pub_key_str, digest_type`*`)`**

  Verifies whether the signature string matches the digest string, and returns 1 or 0 to indicate whether verification succeeded or failed.

  `digest_str` is the digest string. It can be generated by calling **`CREATE_DIGEST()`**. ***`digest_type`***indicates the digest algorithm used to generate the digest string.

  `sig_str` is the signature string. It can be generated by calling **`ASYMMETRIC_SIGN()`**.

  `pub_key_str` is the public key string of the signer. It corresponds to the private key passed to**`ASYMMETRIC_SIGN()`** to generate the signature string and must be a valid key string in PEM format.***`algorithm`*** indicates the encryption algorithm used to create the key.

  Supported ***`algorithm`*** values: **`'RSA'`**, **`'DSA'`**

  Supported ***`digest_type`*** values: **`'SHA224'`**, **`'SHA256'`**, **`'SHA384'`**, **`'SHA512'`**

```
-- Set the encryption algorithm and digest type
```

```
SET @algo = 'RSA';
SET @dig_type = 'SHA224';

-- Create private/public key pair
SET @priv = CREATE_ASYMMETRIC_PRIV_KEY(@algo, 1024);
SET @pub = CREATE_ASYMMETRIC_PUB_KEY(@algo, @priv);

-- Generate digest from string
SET @dig = CREATE_DIGEST(@dig_type, 'The quick brown fox');

-- Generate signature for digest and verify signature against digest
SET @sig = ASYMMETRIC_SIGN(@algo, @dig, @priv, @dig_type);
SET @verf = ASYMMETRIC_VERIFY(@algo, @dig, @sig, @pub, @dig_type);
```

- **`CREATE_ASYMMETRIC_PRIV_KEY(`*`algorithm`*`, {`*`key_len`*`|`*`dh_secret`*`})`**

  Creates a private key using the given algorithm and key length or DH secret, and returns the key as a binary string in PEM format. If key generation fails, the result is **`NULL`**.

  Supported ***`algorithm`*** values: **`'RSA'`**, **`'DSA'`**, **`'DH'`**

  Supported ***`key_len`*** values: The minimum key length in bits is 1024. The maximum key length depends on the algorithm: 16,384 for RSA and 10,000 for DSA. These lengths are constraints imposed by OpenSSL.

  For DH keys, pass a shared DH secret instead of a key length. To create the secret, pass the key length to **`CREATE_DH_PARAMETERS()`**.

  This example creates a 2,048-bit DSA private key, then derives a public key from the private key:

  ```
  SET @priv = CREATE_ASYMMETRIC_PRIV_KEY('DSA', 2048);
  SET @pub = CREATE_ASYMMETRIC_PUB_KEY('DSA', @priv);
  ```

  For an example showing DH key generation, see the description of **`ASYMMETRIC_DERIVE()`**.

  Some general considerations in choosing key lengths and encryption algorithms:

  - The strength of encryption for private and public keys increases with the key size, but the time for key generation increases as well.

  - Generation of DH keys takes much longer than RSA or RSA keys.

  - Asymmetric encryption functions are slower than symmetric functions. If performance is an important factor and the functions are to be used very frequently, you are better off using symmetric encryption. For example, consider using **`AES_ENCRYPT()`** and **`AES_DECRYPT()`**.

- **`CREATE_ASYMMETRIC_PUB_KEY(`*`algorithm`*`, `*`priv_key_str`*`)`**

Derives a public key from the given private key using the given algorithm, and returns the key as a binary string in PEM format. If key derivation fails, the result is `NULL`.

`priv_key_str` must be a valid key string in PEM format. ***`algorithm`*** indicates the encryption algorithm used to create the key.

Supported ***`algorithm`*** values: `'RSA'`, `'DSA'`, `'DH'`

For a usage example, see the description of `CREATE_ASYMMETRIC_PRIV_KEY()`.

- `CREATE_DH_PARAMETERS(`*`key_len`*`)`

  Creates a shared secret for generating a DH private/public key pair and returns a binary string that can be passed to `CREATE_ASYMMETRIC_PRIV_KEY()`. If secret generation fails, the result is null.

  Supported ***`key_len`*** values: The minimum and maximum key lengths in bits are 1024 and 10,000. These lengths are constraints imposed by OpenSSL.

  For an example showing how to use the return value for generating symmetric keys, see the description of `ASYMMETRIC_DERIVE()`.

  ```
  SET @dhp = CREATE_DH_PARAMETERS(1024);
  ```

- `CREATE_DIGEST(`*`digest_type, str`*`)`

  Creates a digest from the given string using the given digest type, and returns the digest as a binary string. If digest generation fails, the result is `NULL`.

  Supported ***`digest_type`*** values: `'SHA224'`, `'SHA256'`, `'SHA384'`, `'SHA512'`

  ```
  SET @dig = CREATE_DIGEST('SHA512', The quick brown fox');
  ```

  The resulting digest string is suitable for use with `ASYMMETRIC_SIGN()` and `ASYMMETRIC_VERIFY()`.