



# PeerSec Cryptographic APIs

<b>Overview</b>	<b>1</b>
Documentation Style Conventions	1
<b>Source Code Notes</b>	<b>2</b>
<b>Package Structure</b>	<b>2</b>
<b>Integer Sizes</b>	<b>2</b>
<b>Configurable Features</b>	<b>2</b>
<b>API Documentation</b>	<b>5</b>
<b>Symmetric Key Ciphers</b>	<b>5</b>
Advanced Encryption Standard (AES)	6
Triple Data Encryption Standard (3DES)	8
SEED	10
ARC4	12
<b>Key Derivation Functions</b>	<b>13</b>
PBKDF2	13
<b>RSA Public Key Cryptography</b>	<b>15</b>
Public Key Extraction (X.509)	16
<i>psX509ParseCertFile</i>	16
<i>psX509FreeCert</i>	17
Private Key Extraction (PKCS#1 and PKCS#8)	18
<i>pkcs1ParsePrivFile</i>	18
<i>pkcs1ParsePrivBin</i>	19
<i>pkcs8ParsePrivBin</i>	20
<i>psFreePubKey</i>	21
RSA Encryption Primitives	22



<i>psRsaCrypt</i>	22
PKCS1-v1_5 Encryption and Signatures	24
<i>psRsaEncryptPub</i>	24
<i>psRsaDecryptPriv</i>	26
<i>psRsaEncryptPriv</i>	27
<i>psRsaDecryptPub</i>	29
OAEP Encryption Scheme	30
<i>pkcs1OaepEncode</i>	30
<i>pkcs1OaepDecode</i>	32
PSS Signature Scheme	34
<i>pkcs1PssEncode</i>	34
<i>pkcs1PssDecode</i>	36
<b>Digest Algorithms</b>	<b>38</b>
Secure Hash Algorithm 2 (SHA-256)	38
Secure Hash Algorithm 1 (SHA-1)	39
Message-Digest Algorithm 5 (MD5)	40

API Version: PSCRYPTO\_3.1

# Overview

The underlying technologies of any security system include the cryptographic primitives that encrypt and authenticate data. Application integrators using PeerSec security protocol products such as **MatrixSSL** and **MatrixSSH** generally do not need to work with this lower level functionality. However, there are many use-cases in which access to the base cryptographic functions are needed. This document is a technical reference for application integrators that wish to use the PeerSec Cryptographic API.

This document is not a general reference for symmetric key ciphers, public-key encryption, or message authentication codes.

## Documentation Style Conventions

- File names and directory paths are *italicized*.
- C code literals are distinguished with this `Monaco` font.

# Source Code Notes

## Package Structure

The public interface function prototypes are defined in the *cryptoApi.h* file. Applications compiling with PeerSec Crypto APIs should only have to include this single header file from their own source.

```
#include "cryptoApi.h"
```

The *cryptoApi.h* file includes other package-specific header files using relative paths based on the default directory structure. All optional product features are enabled and disabled by toggling documented header defines so there should never be a need to restructure the include logic within the header files or to move the header files from the default directory locations.

## Integer Sizes

PeerSec products are designed with the assumption that integer sizes are 32-bit. This is clarified with the use of `int32` and `uint32` type definitions throughout. These are typedefs at the top of the *osdep/osdep.h* header file. This layer enables global redefinitions for platforms that do not support 32-bit integer types as the native `int` type.

## Configurable Features

There is a set of optional features that are available to include or exclude when building the `pscopyto` library. Each of these options are simply pre-processor defines that can be disabled by commenting out the `#define` line in the specified header files. If an API has a dependency on any of these optional features there will be a **Define Dependencies** section in the documentation for that function.

<code>USE_FILE_SYSTEM</code>	<i>coreConfig.h</i>	Enables file access for parsing X.509 certificates and PKCS private keys.
<code>USE_CRYPTTO_TRACE</code>	<i>cryptoConfig.h</i>	Enables internal debug trace
<code>USE_RSA</code>	<i>cryptoConfig.h</i>	Enable to support RSA public key cryptography and PKCS file parsing

USE_DH	<i>cryptoConfig.h</i>	Enable to support Diffie-Hellman public key cryptography and PKCS#3 file parsing
USE_AES	<i>cryptoConfig.h</i>	Enable to support the AES symmetric block cipher
USE_SEED	<i>cryptoConfig.h</i>	Enable to support the SEED symmetric block cipher
USE_3DES	<i>cryptoConfig.h</i>	Enable to support the 3DES symmetric block cipher. This should be defined if private key encryption will be used.
USE_DES	<i>cryptoConfig.h</i>	Enable to support the DES symmetric block cipher. Not recommended
USE_ARC4	<i>cryptoConfig.h</i>	Enable to support the ARC4 symmetric stream cipher
USE_SHA1	<i>cryptoConfig.h</i>	Enable to support the SHA-1 digest algorithm.
USE_MD5	<i>cryptoConfig.h</i>	Enable to support the MD5 digest algorithm.
USE_SHA256	<i>cryptoConfig.h</i>	Enable to support the SHA-256 digest algorithm.
USE_HMAC	<i>cryptoConfig.h</i>	Enable to support the HMAC digest algorithm. Requires either SHA-1 or MD5
USE_X509	<i>cryptoConfig.h</i>	Enable to support X.509 certificate parsing. Requires MD5, SHA-1, and RSA to be enabled.
USE_PRIVATE_KEY_PARSING	<i>cryptoConfig.h</i>	Enable to support the parsing of PKCS#1 formatted RSA private keys. Requires USE_FILE_SYSTEM if parsing from files.
USE_PKCS8	<i>cryptoConfig.h</i>	Enable to support the parsing PKCS#8 formatted RSA private keys. Requires USE_FILE_SYSTEM if parsing from files.
USE_PKCS5	<i>cryptoConfig.h</i>	Enable to support password encryption of private keys and key derivation functions.

USE_PKCS1_OAEP	<i>cryptoConfig.h</i>	Enables the PKCS#1 OAEP encryption scheme for RSA
USE_PKCS1_PSS	<i>cryptoConfig.h</i>	Enables the PKCS#1 PSS signature scheme for RSA

# API Documentation

## Symmetric Key Ciphers

The ciphers documented in this section enable plaintext data to be encrypted by one party and decrypted by another.

A symmetric key cipher is one in which the encryption and decryption keys are identical values. Therefore, it should be obvious that keeping the key a secret between the two parties is essential when using these ciphers to communicate securely. The APIs in this document do not account for how symmetric keys are generated or how these secret keys are shared between the communicating parties.

Symmetric ciphers come in two varieties; **block** or **stream**. The mathematical key mechanism is different between the two types but the practical difference has been summarized in the naming. Block ciphers differ from stream ciphers in that the data is processed a block at a time (often 8 or 16 bytes) rather than as a byte-by-byte stream. Block ciphers are generally recommended over stream ciphers as block ciphers are considered easier to safely implement.

The ciphers in this section are ordered with the most recommended at the top.

## Advanced Encryption Standard (AES)

AES is the faster, stronger successor to the DES block cipher. This 16-byte block cipher supports key sizes of 128-bit, 192-bit, and 256-bit.

The following AES APIs operate in CBC mode (Cipher Block Chaining) to mitigate known issues associated with block ciphers using default ECB mode. The one practical concern CBC mode introduces is the addition and management of a 16-byte Initialization Vector (IV) between the communicating parties.

### Memory Profile

There are no internal memory allocations performed by these functions.

### Define Dependencies

USE\_AES must be defined in *cryptoConfig.h* for access to these functions.

### Function Prototypes and Parameters

```
int32 psAesInit(psCipherContext_t *ctx, unsigned char *IV,
               unsigned char *key, int32 keylen);
```

ctx	Output	Returned cipher context to be used with psAesEncrypt and/or psAesDecrypt
IV	Input	A 16-byte initialization vector used to seed the encryption/decryption of the leading block of data. This value must be the same on each side of the communication channel.
key	Input	The symmetric key. This value must be the same on each side of the communication channel.
keylen	Input	Effective byte length of key. Must be 16, 24, or 32
Return Value	Output	0 on success, < 0 on failure

```
int32 psAesEncrypt(psCipherContext_t *ctx, unsigned char *pt,
    unsigned char *ct, int32 len);
```

ctx	Input	Cipher context returned from a previous call to psAesInit
pt	Input	The plain text data that is to be encrypted
ct	Output	The encrypted cipher text. User must have allocated storage for this parameter.
len	Input	The length in bytes of the plain text (pt) and the output length of the cipher text. Must be a multiple of 16.
Return Value	Output	len value on success, < 0 on failure

```
int32 psAesDecrypt(sslCipherContext_t *ctx, unsigned char *ct,
    unsigned char *pt, int32 len);
```

ctx	Input	Cipher context returned from a previous call to psAesInit
ct	Input	The cipher text data that is to be decrypted
pt	Output	The decrypted plain text. User must have allocated storage for this parameter.
len	Input	The length in bytes of the cipher text (ct) and the output length of the plain text. Must be a multiple of 16.
Return Value	Output	len value on success, < 0 on failure

### Triple Data Encryption Standard (3DES)

3DES is a legacy 8 byte block cipher still in widespread use today. Mathematically, the effective key size is 112 bits but overhead parity bits result in a fixed key size of 24 bytes. Although there are no known severe cryptographic weaknesses in 3DES, the AES block cipher is recommended because of the larger key sizes it supports.

The following 3DES APIs operate in CBC (Cipher Block Chaining) mode to mitigate known issues associated with block ciphers using default ECB mode. The one practical concern CBC mode introduces is the addition and management of an 8 byte Initialization Vector (IV) between the communicating parties.

#### Memory Profile

There are no internal memory allocations performed by these functions.

#### Define Dependencies

USE\_3DES must be defined in *cryptoConfig.h* for access to these functions.

#### Function Prototypes and Parameters

```
int32 psDes3Init(psCipherContext_t *ctx, unsigned char *IV,
                unsigned char *key, int32 keylen);
```

ctx	Output	Returned cipher context to be used with psDes3Encrypt and/or psDes3Decrypt
IV	Input	An 8-byte initialization vector used to seed the encryption/decryption of the leading block of data. This value must be the same on each side of the communication channel.
key	Input	The 24-byte symmetric key. This value must be the same on each side of the communication channel.
keylen	Input	Effective length of key -- Must be 24
Return Value	Output	0 on success, < 0 on failure

```
int32 psDes3Encrypt(psCipherContext_t *ctx, unsigned char *pt,
    unsigned char *ct, int32 len);
```

ctx	Input	Cipher context returned from a previous call to psDes3Init
pt	Input	The plain text data that is to be encrypted
ct	Output	The encrypted cipher text. User must have allocated storage for this parameter.
len	Input	The length in bytes of the plain text (pt) and the output length of the cipher text. Must be a multiple of 8.
Return Value	Output	len value on success, < 0 on failure

```
int32 psDes3Decrypt(psCipherContext_t *ctx, unsigned char *ct,
    unsigned char *pt, int32 len);
```

ctx	Input	Cipher context returned from a previous call to psDes3Init
ct	Input	The cipher text data that is to be decrypted
pt	Output	The decrypted plain text. User must have allocated storage for this parameter.
len	Input	The length in bytes of the cipher text (ct) and the output length of the plain text. Must be a multiple of 8.
Return Value	Output	len value on success, < 0 on failure

## SEED

SEED is an alternative 16-byte block cipher supporting 128-bit key lengths.

The following SEED APIs operate in CBC mode (Cipher Block Chaining) to mitigate known issues associated with block ciphers using default ECB mode. The one practical concern CBC mode introduces is the addition and management of a 16-byte Initialization Vector (IV) between the communicating parties.

### Memory Profile

There are no internal memory allocations performed by these functions.

### Define Dependencies

USE\_SEED must be defined in *cryptoConfig.h* for access to these functions.

### Function Prototypes and Parameters

```
int32 psSeedInit(psCipherContext_t *ctx, unsigned char *IV,
                unsigned char *key, int32 keylen);
```

ctx	Output	Returned cipher context to be used with psSeedEncrypt and/or psSeedDecrypt
IV	Input	A 16-byte initialization vector used to seed the encryption/decryption of the leading block of data. This value must be the same on each side of the communication channel.
key	Input	The 16-byte symmetric key. This value must be the same on each side of the communication channel.
keylen	Input	Effective length of key -- Must be 16
Return Value	Output	0 on success, < 0 on failure

```
int32 psSeedEncrypt(sslCipherContext_t *ctx, unsigned char *pt,
    unsigned char *ct, int32 len);
```

ctx	Input	Cipher context returned from a previous call to psSeedInit
pt	Input	The plain text data that is to be encrypted
ct	Output	The encrypted cipher text. User must have allocated storage for this parameter.
len	Input	The length in bytes of the plain text (pt) and the output length of the cipher text. Must be a multiple of 16.
Return Value	Output	len value on success, < 0 on failure

```
int32 psSeedDecrypt(sslCipherContext_t *ctx, unsigned char *ct,
    unsigned char *pt, int32 len);
```

ctx	Input	Cipher context returned from a previous call to psSeedInit
ct	Input	The cipher text data that is to be decrypted
pt	Output	The decrypted plain text. User must have allocated storage for this parameter.
len	Input	The length in bytes of the cipher text (ct) and the output length of the plain text. Must be a multiple of 16.
Return Value	Output	len value on success, < 0 on failure

## ARC4

ARC4 (Alleged RC4) is the most widely adopted stream cipher. This cipher works on variable length keys with 128-bit being the most popular size. Key sizes may range from 8-bits to 2048-bits. A single function is provided for either encryption or decryption. There are some security weaknesses with ARC4, however it is very fast and uses very little code space.

### Memory Profile

There are no internal memory allocations performed by these functions.

### Define Dependencies

USE\_ARC4 must be defined in *cryptoConfig.h* for access to these functions.

### Function Prototypes and Parameters

```
void psArc4Init(psCipherContext_t *ctx, unsigned char *key,
               int32 keylen);
```

ctx	Output	Returned cipher context to be used with psArc4
key	Input	The variable length symmetric key. This value must be the same on each side of the communication channel.
keylen	Input	Byte length of key -- Must be between 1 and 256

```
int32 psArc4(psCipherContext_t *ctx, unsigned char *in,
             unsigned char *out, int32 len);
```

ctx	Input	Cipher context returned from a previous call to psArc4Init
in	Input	The data that is to be processed (encrypt or decrypt)
out	Output	The output processed data (encrypted or decrypted). User must have allocated storage for this parameter.
len	Input	The length in bytes of the data
Return Value	Output	len value on success, < 0 on failure

## Key Derivation Functions

Using a plaintext password directly as a symmetric encryption key is prone to brute force and dictionary password attacks. The addition of a salt value “mixed in” with the password reduces the effectiveness of a dictionary attack. A salt value is an arbitrary byte sequence that makes the search space for dictionary attacks much larger. It is not a secret value, and often held in plaintext with the object that is encrypted. For example, in plain text as the first few bytes of an encrypted file.

Making the “mixing in” of the password and salt computationally expensive further reduces the effectiveness of brute force password attacks. A key derivation function combines a password with a salt value in a computationally expensive way (typically using a one way hash, such as SHA-1), producing an arbitrary number of bytes that can be used as input to a symmetric cryptography algorithm such as AES.

### PBKDF2

Password Based Key Derivation Function 2 is part of the PKCS #5 specification, and recommended for generation of strong symmetric keys using a password or passphrase. PBKDF1 is an older version of this function and included for legacy protocols that specify it.

### Memory Profile

There are no internal memory allocations performed by these functions.

### Define Dependencies

USE\_PKCS5, USE\_HMAC and USE\_SHA must be defined in *cryptoConfig.h* for access to these functions.

### Function Prototypes and Parameters

```
void pkcs5pbkdf2(unsigned char *password, uint32 pLen,  
                unsigned char *salt, uint32 sLen, int32 rounds,  
                unsigned char *key, uint32 kLen);
```

password	Input	Password or passphrase to use.
pLen	Input	Length of password in bytes
salt	Input	Salt value

sLen	Input	Length of salt in bytes, recommended 8 bytes minimum.
rounds	Input	Number of iterations for the HMAC-SHA mixing function. More rounds makes the key generation slower, but also makes the brute force attacks take longer. 1024 iterations is the recommended minimum, with 4096 being a common default value.
key	Output	Memory location to write kLen bytes of the generated key.
kLen	Input	Requested length of generated key in bytes. Typically this is the number of bytes required for the key of the symmetric encryption algorithm being used.
Return Value	void	void

## RSA Public Key Cryptography

Public key cryptography differs from symmetric key cryptography in that the communicating parties do not use identical key values when working with data. In public key cryptography one side is said to hold the public key and the other to hold the private key. The primary benefit to this method is that the public key is freely available so two entities may communicate securely without ever having agreed on a single common key. The real world use-case most readily understood is that of a person using a Web browser to securely purchase products online. The server of that transaction can prove its identity to the user by the fact it holds the private key.

Public key cryptography is not a good option for the generic encryption and decryption of plain text information for a couple reasons. First, public key operations are quite slow in comparison to symmetric key encryption. Second, the maximum data length in public key operations can not exceed the key length (typically 128-bytes or 256-bytes). For these reasons, the relationship between private and public keys have resulted in two common usages; **public key encryption** and **digital signatures**.

Public key encryption is the mechanism by which the public key holder encrypts data that it wants to exchange with the holder of the private key. Frequently, the data being exchanged is a symmetric key that the two parties will use to encrypt and decrypt traffic going forward. If the recipient of this message is able to decrypt the symmetric key and use it to communicate with the sender, the public key holder can be confident the other side has possession of the private key, thus proving its identity.

Digital signatures work the opposite direction. The holder of the private key encrypts data into a signature. If the public key holder can decrypt the signature and verify the contents independently it can be confident the private key holder was the signer and the data has not been compromised.

## Public Key Extraction (X.509)

Although it is possible, public keys are generally not stored as standalone streams of bytes like symmetric keys often are. This is because public keys are often intended to have a very long useful life (years) and so they must be accompanied by the information that went into creating them such as who issued the key and what the key is to be used for. The collection of public key and related information are stored in **certificates**.

X.509 is the standard that defines the formats for public key certificates. Once in the X.509 specification format the byte stream may be left in the raw ASN.1 binary form or it may be encoded in a textual base-64 format as a PEM file. PEM files are easily identified by the enclosing -----BEGIN <FILE\_TYPE>----- and -----END <FILE\_TYPE>----- tags.

For the purposes of this cryptography document, the X.509 parsing routine will only be discussed in terms of retrieving the public key material for use in the lower level encryption routines that follow.

### psX509ParseCertFile

#### Prototype

```
int32 psX509ParseCertFile(psPool_t *pool, char *certFile,
    psX509Cert_t **outcert, int32 flags);
```

#### Parameters

pool	Input	Memory pool for internal allocations. NULL for default system memory allocation.
certFile	Input	File name of PEM encoded certificate(s)
outcert	Output	Structure containing the parsed certificate information.
flags	Input	0 or CERT_STORE_UNPARSED_BUFFER, CERT_STORE_DN_BUFFER

### Return Values

PS_SUCCESS	Successful parse
PS_FAILURE PS_LIMIT_FAIL	X.509 parse failure. Enable pscrypto trace for more information.
PS_ARG_FAIL	Failure on bad input parameters.
PS_MEM_FAIL	Failure on internal memory allocation
PS_PLATFORM_FAIL	Failure to locate or open certFile
PS_UNSUPPORTED_FAIL	Failure on parse of unsupported X.509 algorithm

### Description

This routine is used to convert X.509 PEM formatted certificate files into a psX509Cert\_t data type (as defined in *x509.h*).

The certificate public key will be accessible in the publicKey member of the structure as a psPubKey\_t data type.

### Memory Profile

Caller must free outcert using psX509FreeCert if function returns successfully

### Define Dependencies

The user must ensure USE\_X509 and USE\_FILE\_SYSTEM are enabled in the configuration header files.

## psX509FreeCert

### Prototype

```
void psX509FreeCert(psX509Cert_t *cert);
```

### Parameters

cert	Input	Structure to free
------	-------	-------------------

### Description

Free an psX509Cert\_t structure that was allocated by a previous call to psX509ParseCertFile.

## Private Key Extraction (PKCS#1 and PKCS#8)

Private keys are typically stored as a key stream and are often password protected. The PKCS#1 standard is specific to RSA. The PKCS#8 standard is a more generic public key specification and is not as widely deployed.

### pkcs1ParsePrivFile

#### Prototype

```
int32 pkcs1ParsePrivFile(psPool_t *pool, char *privFile,
    char *password, psPubKey_t **key);
```

#### Parameters

pool	Input	Memory pool for internal allocations. NULL for default system memory allocation (or if using open source product version)
privFile	Input	PKCS#1 PEM formatted private key file
password	Input	String password for the privFile. If private key file is not password protected this parameter must be NULL.
key	Output	The private key material.

#### Return Values

PS_SUCCESS	Successful parse
PS_ARG_FAIL	Failure due to bad input parameters
PS_PLATFORM_FAIL	Failed to locate or unable to open privFile
PS_MEM_FAIL	Failure during internal memory allocation
PS_PARSE_FAIL	Failure during key parse
PS_UNSUPPORTED_FAIL	Failed due to unsupported algorithm in key file

**Description**

This routine is used to extract private key material from a PKCS#1 PEM formatted file.

The supported password encryption standard is PKCS#5 PBKDF1 DES-EDE3-CBC.

**Memory Profile**

The key parameter must be freed by the user using `psFreePubKey` if the function returns `PS_SUCCESS`.

**Define Dependencies**

The user must ensure `USE_PRIVATE_KEY_PARSING` and `USE_FILE_SYSTEM` are enabled in the configuration header files. If the private key files are password encrypted (recommended) `USE_PKCS5` must be enabled.

`pkcs1ParsePrivBin`

**Prototype**

```
int32 pkcs1ParsePrivBin(psPool_t *pool, unsigned char *keyBuf,
    int32 keyBufLen, psPubKey_t **key);
```

**Parameters**

<code>pool</code>	Input	Memory pool for internal allocations. NULL for default system memory allocation (or if using open source product version)
<code>keyBuf</code>	Input	ASN.1 DER private key stream in PKCS#1 format
<code>keyBufLen</code>	Input	Byte length of the <code>keyBuf</code> parameter
<code>key</code>	Output	Internally allocated <code>psPubKey_t</code> formatted private key for use in lower level RSA routines

**Return Values**

<code>PS_SUCCESS</code>	Successful parse
<code>PS_MEM_FAIL</code>	Failure on internal memory allocation

PS_PARSE_FAIL	Failure during key parse
---------------	--------------------------

### Description

This routine is used to extract private key material from a PKCS#1 ASN.1 encoded key stream. There are no password encryption options at this level of the standard.

### Memory Profile

The key parameter must be freed by the user using psFreePubKey if the function returns PS\_SUCCESS.

### Define Dependencies

The user must ensure USE\_PRIVATE\_KEY\_PARSING is enabled in the configuration header file.

## pkcs8ParsePrivBin

### Prototype

```
int32 pkcs8ParsePrivBin(psPool_t *pool, unsigned char *keyBuf,
    int32 keyBufLen, char *password, psPubKey_t **key);
```

### Parameters

pool	Input	Memory pool for internal allocations. NULL for default system memory allocation (or if using open source product version)
keyBuf	Input	ASN.1 DER private key stream in PKCS#8 format
keyBufLen	Input	Byte length of the keyBuf parameter
password	Input	NULL if unencrypted or password if encrypted.
key	Output	Internally allocated psPubKey_t formatted private key for use in lower level RSA routines

### Return Values

PS_SUCCESS	Successful parse
------------	------------------

PS_UNSUPPORTED_FAIL	Failed due to unsupported algorithm in key file
PS_FAILURE	Failure during key parse

**Description**

This routine is used to extract private key material from a PKCS#8 ASN.1 encoded key stream.

The supported password encryption standard is PKCS#5 PBKDF2 DES-EDE3-CBC.

**Memory Profile**

The key parameter must be freed by the user using psFreePubKey if the function returns PS\_SUCCESS.

**Define Dependencies**

The user must ensure USE\_PRIVATE\_KEY\_PARSING and USE\_PKCS8 are enabled in the configuration header files. If the private key files are password encrypted (recommended) USE\_PKCS5 must be enabled.

[psFreePubKey](#)

**Prototype**

```
void psFreePub(psPubKey_t *key);
```

**Parameters**

key	Input	Structure to free
-----	-------	-------------------

**Description**

Free a psPubKey\_t structure that was allocated by a previous call to pkcs1ParsePrivBin, pkcs1ParsePrivFile or pkcs8ParsePrivBin.

## RSA Encryption Primitives

An RSA operation, whether it be encryption or a digital signature, generally involves two steps. The first is the **RSA primitive** which is responsible for the low level transition between the plaintext data and the cipher-text data. The second is the **encoding scheme** that formats the plaintext data. Sometimes the encoding scheme is referred to as the padding scheme or just “padding”.

The following routine is the RSA primitive for use-cases in which an encoding scheme is being performed separately.

Note that the key parsing functions from the previous sections have been working with `psPubKey_t` structures. The RSA specific cryptographic functions in the following sections reference the `psRsaKey_t` structure. Use the `key` member of the `psPubKey_t` data type for access to the `psRsaKey_t` structure.

### psRsaCrypt

#### Prototype

```
int32 psRsaCrypt(psPool_t *pool, unsigned char *in, uint32 inLen,
                unsigned char *out, uint32 *outLen), psRsaKey_t *key, int32 type);
```

#### Parameters

<code>pool</code>	Input	Memory pool for internal allocations. NULL for default system memory allocation
<code>in</code>	Input	The data to perform the RSA operation on
<code>inLen</code>	Input	Byte length of the <code>in</code> buffer. Must be the length of the RSA key being used.
<code>out</code>	Output	Buffer to hold the output message. May be the same location as the <code>in</code> buffer.
<code>outLen</code>	Input/Output	As input, the allocated byte length of the <code>out</code> parameter. As output, the length of the encrypted message written to <code>out</code> .
<code>key</code>	Input	The RSA key
<code>type</code>	Input	Either <code>PRIVKEY_TYPE</code> or <code>PUBKEY_TYPE</code>

**Return Values**

PS_SUCCESS	Success.
PS_ARG_FAIL	Failure due to bad input parameters
PS_FAILURE	Failure during RSA encryption

**Description**

This is the RSA primitive that performs the transition between ciphertext and plaintext and vice versa. The data lengths for the incoming and outgoing data must exactly match the byte length of the RSA key being used. This is a limitation of RSA cryptography.

This function will be required if using OAEP or PSS encoding schemes as described below. This function is not necessary when using the RSA PKCS1-v1\_5 encoding scheme functions.

**Define Dependencies**

The user must ensure USE\_RSA is enabled in the configuration header file.

## PKCS1-v1\_5 Encryption and Signatures

The PeerSec cryptographic layer was originally created for use within the SSL protocol. As such, the encryption and signature schemes that are used by default are based on PKCS1v1\_5. These following four routines combine PKCS1v1\_5 padding and the RSA primitive so it is not necessary to use the `psRsaCrypt` routine when using this interface. For OAEP and PSS schemes, see the sections that follow.

### psRsaEncryptPub

#### Prototype

```
int32 psRsaEncryptPub(psPool_t *pool, psRsaKey_t *key,
    unsigned char *in, int32 inLen,
    unsigned char *out, int32 outLen);
```

#### Parameters

pool	Input	Memory pool for internal allocations. NULL for default system memory allocation
key	Input	RSA public key
in	Input	The buffer to encrypt
inLen	Input	Byte length of the buffer to encrypt. Must be smaller than the RSA key size
out	Output	Buffer to hold the output encrypted message
outLen	Input	Must be the byte size of the RSA key

#### Return Values

> 0	Successful encrypt. Returns the key size (encrypted data length)
PS_ARG_FAIL	Failure due to bad input parameters
PS_FAILURE	Failure during RSA encryption

**Description**

Perform an RSA public encryption operation on some data. The corresponding decryption routine is `psRsaDecryptPriv`.

The input data is padded using the PKCS1-v1\_5 standard so the data input length must be several bytes smaller than the RSA key size. The key size can be determined by looking at the `size` member of the `sslRsaKey_t` structure.

The encrypted output will always match the key size and the `outLen` parameter must match this size. Storage for the `out` parameter must be allocated by the user prior to calling this routine.

**Define Dependencies**

The user must ensure `USE_RSA` is enabled in the configuration header file.

## psRsaDecryptPriv

### Prototype

```
int32 psRsaDecryptPriv(psPool_t *pool, psRsaKey_t *key,
    unsigned char *in, int32 inLen,
    unsigned char *out, int32 outLen);
```

### Parameters

pool	Input	Memory pool for internal allocations. NULL for default system memory allocation
key	Input	RSA private key
in	Input	The buffer to decrypt
inLen	Input	Byte length of the buffer to decrypt. Must match the RSA key size.
out	Output	Buffer to hold the output decrypted message
outLen	Input	Must be the known size of the unencrypted data

### Return Values

> 0	Successful decrypt. Returns the size of the decrypted data (outLen value)
PS_ARG_FAIL	Failure due to bad input parameters
PS_FAILURE	Failure during RSA decryption

### Description

Perform an RSA private decryption operation on some data. The corresponding encryption routine is `psRsaEncryptPub`. The decrypted data will be unpadding using PKCS1-v1\_5.

The caller must know the decrypted output length in bytes prior to calling this routine. Storage for the out parameter must be allocated by the user prior to calling this routine.

### Define Dependencies

The user must ensure `USE_RSA` is enabled in the configuration header file.

## psRsaEncryptPriv

### Prototype

```
int32 psRsaEncryptPriv(psPool_t *pool, psRsaKey_t *key,
    unsigned char *in, int32 inLen,
    unsigned char *out, int32 outLen);
```

### Parameters

pool	Input	Memory pool for internal allocations. NULL for default system memory allocation.
key	Input	RSA private key
in	Input	The buffer to encrypt
inLen	Input	Byte length of the buffer to encrypt. Must be smaller than the RSA key size
out	Output	Buffer to hold the output encrypted message
outLen	Input	Must be the byte size of the RSA key

### Return Values

> 0	Successful encrypt. Returns the key size (encrypted data length)
PS_ARG_FAIL	Failure due to bad input parameters
PS_FAILURE	Failure during RSA encryption

### Description

Perform an RSA private encryption operation on some data. This is commonly referred to as a digital signature. The corresponding decryption routine is psRsaDecryptPub.

The input data is padded using the PKCS1-v1\_5 standard and so the data input length must be several bytes smaller than the RSA key size. The key size can be determined by looking at the size member of the psRsaKey\_t structure.



The encrypted output will always match the key size and the outLen parameter must match this size. Storage for the out parameter must be allocated by the user prior to calling this routine.

**Define Dependencies**

The user must ensure USE\_RSA is enabled in the configuration header file.

## psRsaDecryptPub

### Prototype

```
int32 psRsaDecryptPub(psPool_t *pool, psRsaKey_t *key,
    unsigned char *in, int32 inLen,
    unsigned char *out, int32 outLen);
```

### Parameters

pool	Input	Memory pool for internal allocations. NULL for default system memory allocation
key	Input	RSA private key
in	Input	The buffer to decrypt
inLen	Input	Byte length of the buffer to decrypt. Must match the RSA key size.
out	Output	Buffer to hold the output decrypted message
outLen	Input	Must be the known size of the unencrypted data

### Return Values

> 0	Successful decrypt. Returns the size of the decrypted data (outLen value)
PS_ARG_FAIL	Failure due to bad input parameters
PS_FAILURE	Failure during RSA decryption

### Description

Perform an RSA public decryption operation on some data. This routine is used to validate a digital signature. The corresponding encryption routine is psRsaEncryptPriv.

The caller must know the decrypted output length in bytes prior to calling this routine. The decrypted data will be unpadding using PKCS1-v1\_5. Storage for the out parameter must be allocated by the user prior to calling this routine.

### Define Dependencies

The user must ensure USE\_RSA is enabled in the configuration header file.

## OAEP Encryption Scheme

Unlike the PKCS1-v1\_5 encoding scheme that can be used for both encryption and signature operations, the OAEP scheme may only be used for encryption operations. That is, encryption with a public key and decryption with a private key. For the full RSA encryption operation these OAEP routines must be used in conjunction with the `psRsaCrypt` primitive.

### pkcs1OaepEncode

#### Prototype

```
int32 pkcs10aepEncode(psPool_t *pool, const unsigned char *msg,
    uint32 msgLen, const unsigned char *label, uint32 labelLen,
    unsigned char *seed, uint32 seedLen, uint32 modulusBitLen,
    int32 hashIndex, unsigned char *out, uint32 *outLen);
```

#### Parameters

pool	Input	Memory pool for internal allocations. NULL for default system memory allocation
msg	Input	The plaintext message on which to perform the encoding
msgLen	Input	The byte length of the msg parameter.
label	Input	Optional label to associate with the message. May be NULL
labelLen	Input	Byte length of the label parameter
seed	Input	NULL (Reserved for testing)
seedLen	Input	0 (Reserved for testing)
modulusBitLen	Input	The RSA key size (in bits) for the key that will perform the RSA primitive encoding operation
hashIndex	Input	0 for SHA1 hash or 1 for MD5 hash
out	Output	The OAEP encoded output ready to be passed to <code>psRsaCrypt</code>
outLen	Input/Output	As input, the allocated byte length of the out parameter. As output, the length of the encrypted message written to out.

### Return Values

PS_SUCCESS	Successful encode. Returns the size of the decrypted data (outLen value)
PS_ARG_FAIL	Failure due to bad input parameters
PS_FAILURE	Encode failure
PS_LIMIT_FAIL	Message was too large
PS_MEM_FAIL	Internal memory allocation failure
PS_PLATFORM_FAIL	Error in psGetEntropy call

### Description

Optimal Asymmetric Encryption Padding (OAEP) is a stronger padding scheme than PKCS1-v1\_5 and should be used in new implementations.. This encoding routine is used on the encryption side and will generate data that should be passed to psRsaCrypt for PUBKEY\_TYPE encryption.

The msg being encoded has a maximum length constraint of  $k - 2hLen - 2$  octets, where  $k$  is the octet length of the RSA key size (128 for a 1024-bit for exaple) and  $hLen$  is the length of the hash function being used (20 for SHA1 or 16 for MD5). So, if a 1024-bit key is being used with a SHA1 hash the maximum message size would be 86 bytes.

The label parameter is an optional string value that will be incorporated into the hash operations of the padding.

The OAEP padding scheme does not restrict the underlying hash algorithm that may be used. SHA1 and MD5 are supported in this version of the implementation. The hashIndex is the index into the psHashList array where SHA1 is the first (0) and MD5 is the second (1). SHA1 is the recommended hash function.

On a successful encode, the output data will be the byte length of the RSA key size and the out parameter must be allocated to be, at least, that large.

### Define Dependencies

The user must ensure USE\_RSA and USE\_PKCS1\_OEAP are enabled in the configuration header file.

## pkcs1OaepDecode

### Prototype

```
int32 pkcs1OaepDecode(psPool_t *pool, const unsigned char *msg,
    uint32 msglen, const unsigned char *label, uint32 labellen,
    uint32 modulusBitlen, int32 hashIndex, unsigned char *out,
    uint32 *outlen);
```

### Parameters

pool	Input	Memory pool for internal allocations. NULL for default system memory allocation
msg	Input	The plaintext message on which to perform the decoding
msglen	Input	The byte length of the msg parameter.
label	Input	Optional label to associate with the message. May be NULL
labellen	Input	Byte length of the label parameter
modulusBitlen	Input	The RSA key size (in bits) for the key that performed the RSA primitive decoding operation
hashIndex	Input	0 for SHA1 hash or 1 for MD5 hash
out	Output	The OAEP decoded output
outlen	Input/Output	As input, the allocated byte length of the out parameter. As output, the length of the decoded message written to out.

### Return Values

PS_SUCCESS	Successful encode. Returns the size of the decrypted data (outLen value)
PS_ARG_FAIL	Failure due to bad input parameters
PS_FAILURE	Decode failure

PS_LIMIT_FAIL	The msgLen did not match the modulus length of the RSA key
PS_MEM_FAIL	Internal memory allocation failure

**Description**

The decode counterpart to the pkcs10aepEncode routine. A full RSA decryption operation with this function would be to first call the psRsaCrypt primitive in PRIVKEY\_TYPE mode and then to call this function to unpad the data to the final plaintext message.

The msg to decode should necessarily be the exact octet size as the RSA key being used since that input is coming from an RSA decryption.

If a label was used on the encoding operation, that same string label must be used in this decoding operation.

The hashIndex will either be SHA1 (0) or MD5 (1) and also must match whatever hash algorithm was used by the encoding side. SHA1 is recommended.

**Define Dependencies**

The user must ensure USE\_RSA and USE\_PKCS1\_OEAP are enabled in the configuration header file.

## PSS Signature Scheme

Unlike the PKCS1-v1\_5 encoding scheme that is used for both encryption and signature operations, the PSS scheme may only be used for digital signature operations. That is, encryption with a private key and decryption with a public key.

For the full RSA signature operation these PSS routines must be used in conjunction with the psRsaCrypt primitive.

### pkcs1PssEncode

#### Prototype

```
int32 pkcs1PssEncode(psPool_t *pool, const unsigned char *msgHash,
    uint32 msgHashlen, unsigned char *salt, uint32 saltLen,
    int32 hashIndex, uint32 modulusBitlen, unsigned char *out,
    uint32 *outlen);
```

#### Parameters

pool	Input	Memory pool for internal allocations. NULL for default system memory allocation
msgHash	Input	The hash of the plaintext message that is being signed
msgHashlen	Input	The length of msgHash (20 for SHA1 or 16 for MD5, for example)
salt	Input	NULL (reserved for testing)
saltLen	Input	The desired byte length of the random salt data that will be used in the encoding
hashIndex	Input	0 for SHA1 hash or 1 for MD5 hash. Should be the same as the algorithm that hashed msgHash
modulusBitlen	Input	The RSA key size (in bits) for the key that will perform the RSA primitive encoding operation
out	Output	The OAEP encoded output ready to be passed to psRsaCrypt
outlen	Input/Output	As input, the allocated byte length of the out parameter. As output, the length of the encrypted message written to out.

### Return Values

PS_SUCCESS	Successful encode. Returns the size of the decrypted data (outLen value)
PS_ARG_FAIL	Failure due to bad input parameters
PS_FAILURE	Failure encoding
PS_LIMIT_FAIL	Outlen is too small
PS_MEM_FAIL	Internal memory allocation failure
PS_PLATFORM_FAIL	Error in psGetEntropy call

### Description

The Probabilistic Signature Scheme (PSS) is a stronger padding scheme than PKCS1-v1\_5 and should be used in new implementations. This is the padding scheme for the signature creation operation and the output from this function will be passed to psRsaCrypt in a PRIVKEY\_TYPE mode to produce the final ciphertext.

This routine does not perform the actual message hashing of the plaintext. This function accepts the msgHash of the original message to be PSS encoded. The msgHash should either be a SHA1 or MD5 hash and so the msgHashLen will be 20 or 16, respectively.

The saltLen is the number of random bytes that will be incorporated into the padding algorithm and must not be too large. The test used for length limitations is that the modulus byte length of the RSA key being used must be less than msgHashLen + saltLen + 2.

The hashIndex identifies the algorithm to use in the PSS encoding process and the PKCS#1 recommendation is that this algorithm match that which was used to generate the msgHash initially. A value of 0 identifies SHA1 and a value of 1 identifies MD5.

On success, the out output will have an outLen identical to the octet key size of the RSA key that will be used to encrypt.

### Define Dependencies

The user must ensure USE\_RSA and USE\_PKCS1\_PSS are enabled in the configuration header file.

## pkcs1PssDecode

### Prototype

```
int32 pkcs1PssDecode(psPool_t *pool, const unsigned char *msgHash,
    uint32 msgHashlen, const unsigned char *sig, uint32 sigLen,
    uint32 saltlen, int32 hashIndex, uint32 modulusBitlen,
    int32 *result);
```

### Parameters

pool	Input	Memory pool for internal allocations. NULL for default system memory allocation
msgHash	Input	The hash value to verify against
msgHashlen	Input	The length of the msgHash
sig	Input	The signature value to decode.
sigLen	Input	Byte length of the sig parameter
saltlen	Input	The byte length of the random salt value that was used in the encoding portion.
hashIndex	Input	0 for SHA1 hash or 1 for MD5 hash
modulusBitlen	Input	The RSA key size (in bits) for the key that performed the RSA primitive decoding operation
result	Output	Indication of the success or failure of the verification.

### Return Values

PS_SUCCESS	Successful function execution. <b>However, the actual result of the verification is in the result parameter.</b>
PS_ARG_FAIL	Failure due to bad input parameters
PS_FAILURE	Failure during decode
PS_MEM_FAIL	Internal memory allocation failure

### **Description**

The decode counterpart to the `pkcs1PssEncode` routine. A full RSA signature verification operation with this function would be to first call the `psRsaCrypt` primitive in `PUBKEY_TYPE` mode and then to call this function to unpad the data and compare it to the known message hash.

The `msgHash` and `msgHashLen` are the known hash that will be tested against.

The `sig` and `sigLen` parameters identify the decrypted data from the previous `psRsaCrypt` call that need to be decoded.

If a `saltLen` was used on the encoding operation, that same byte length must be used in this decoding operation.

The `hashIndex` will either be SHA1 (0) or MD5 (1) and also must match whatever hash algorithm was used by the encoding side. SHA1 is recommended.

The `result` parameter is the actual signature validation result. A result value of 0 indicates that the signature did not validate. A value of 1 (`PS_TRUE`) indicates the decoded `sig` matches the `msgHash` value.

### **Define Dependencies**

The user must ensure `USE_RSA` and `USE_PKCS1_PSS` are enabled in the configuration header file.

## Digest Algorithms

A digest algorithm is a one-way cryptographic operation that produces a single, short hash value from the controlled inputs. The values are used to validate that the data being acted upon has not been compromised between the sender and receiver. The receiver should be able to run the input data through an independent digest operation and produce the identical code as the sender. Of course, that hash must be exchanged securely in order to prevent the obvious attack of a middle-man simply sending the correct hash along with the compromised data. To combat this problem, public key encryption mechanisms are often used to exchange message digests.

The algorithms are listed with the most recommended appearing first.

### Secure Hash Algorithm 2 (SHA-256)

SHA-256 is among the more common SHA-2 message digest algorithms. It produces a 256-bit digest.

#### Function Prototypes and Parameters

```
void psSha256Init(psDigestContext_t *ctx);
```

ctx	Input	Context structure to be initialized
-----	-------	-------------------------------------

```
void psSha256Update(psDigestContext_t *ctx, unsigned char *buf,
    uint32 len);
```

ctx	Input	Cipher context initialized from a previous call to psSha256Init
buf	Input	The message data to be processed
len	Input	The length in bytes of buf

```
int32 psSha256Final(psDigestContext_t *ctx, unsigned char *hash);
```

ctx	Input	Cipher context used for previous calls to psSha256Update
hash	Output	The digest hash of the data passed into psSha256Update. User must have allocated SHA256_HASH_SIZE bytes of storage for the hash parameter
Return Value	Output	< 0 on error. SHA256_HASH_SIZE on success

### Description

The psSha256Update function may be called any number of times for as much data that requires hashing. The call to psSha256Final will output the digest when the user is done with calls to update.

### Define Dependencies

The user must ensure USE\_SHA256 is enabled in the configuration header file.

## Secure Hash Algorithm 1 (SHA-1)

SHA-1 is a widely used message digest algorithm and cryptographic hash function. It produces a 160-bit digest.

### Function Prototypes and Parameters

```
void psSha1Init(psDigestContext_t *ctx);
```

ctx	Input	Context structure to be initialized
-----	-------	-------------------------------------

```
void psSha1Update(psDigestContext_t *ctx, unsigned char *buf,
    uint32 len);
```

ctx	Input	Cipher context initialized from a previous call to psSha1Init
buf	Input	The message data to be processed

len	Input	The length in bytes of buf
-----	-------	----------------------------

```
int32 psSha1Final(psDigestContext_t *ctx, unsigned char *hash);
```

ctx	Input	Cipher context used for previous calls to psSha1Update
hash	Output	The digest hash of the data passed into psSha1Update. User must have allocated SHA1_HASH_SIZE bytes of storage for the hash parameter
Return Value	Output	< 0 on error. SHA1_HASH_SIZE on success

### Description

The psSha1Update function may be called any number of times for as much data that requires hashing. The call to psSha1Final will output the digest when the user is done with calls to update.

### Define Dependencies

The user must ensure USE\_SHA1 is enabled in the configuration header file.

## Message-Digest Algorithm 5 (MD5)

MD5 is a widely used message digest algorithm. It produces a 128-bit digest.

### Function Prototypes and Parameters

```
void psMd5Init(psDigestContext_t *ctx);
```

ctx	Input	Context structure to be initialized
-----	-------	-------------------------------------

```
void psMd5Update(psDigestContext_t *ctx, unsigned char *buf,
                uint32 len);
```

ctx	Input	Cipher context initialized from a previous call to psMd5Init
buf	Input	The message data to be processed
len	Input	The length in bytes of buf

```
int32 psMd5Final(psDigestContext_t *ctx, unsigned char *hash);
```

ctx	Input	Cipher context used for previous calls to psMd5Update
hash	Output	The digest hash of the data passed into matrixMd5Update. User must have allocated SSL_MD5_HASH_SIZE bytes of storage for the hash parameter
Return Value	Output	< 0 on error. MD5_HASH_SIZE on success

### Description

The psMd5Update function may be called any number of times for as much data that requires hashing. The call to psMd5Final will output the digest when the user is done with calls to update.

### Define Dependencies

The user must ensure USE\_MD5 is enabled in the configuration header file.