

Arm

# DDS Security Plugins Internal APIs Specification

Arm Robotics Software

<b>1 OVERVIEW.....</b>	<b>1</b>
1.1 INTRODUCTION .....	1
1.2 REFERENCES.....	3
1.3 REVISION HISTORY .....	4
1.4 TERMS AND DEFINITIONS .....	5
<b>2 GENERIC OPERATIONS OF FILES AND FORMATS.....</b>	<b>6</b>
2.1 DATA TYPES.....	6
2.1.1 SEC_IO.....	6
2.1.2 ASN1_OBJECT.....	6
2.1.3 BUF_MEM .....	6
2.2 GENERIC OPERATIONS .....	7
2.2.1 Create SEC_IO for a File .....	8
2.2.2 Create SEC_IO for a Memory Buffer.....	8
2.2.3 Create SEC_IO with an Existing Buffer .....	8
2.2.4 Free SEC_IO.....	9
2.2.5 Open File for Reading.....	10
2.2.6 Read ASN1 Object into SEC_IO Memory Buffer.....	11
2.2.7 Set I/O Stream Close Status .....	12
2.2.8 BUF_MEM Allocation.....	13
2.2.9 BUF_MEM Free.....	13
2.2.10 Get the Pointer of BUF_MEM.....	13
<b>3 GENERIC ASYMMETRIC KEY OPERATION .....</b>	<b>14</b>
3.1 DATA TYPES.....	14
3.1.1 Private Key Handle .....	14
3.1.2 Public Key Handle.....	14
3.2 GENERIC FUNCTIONS.....	15
3.2.1 Create Private Key .....	16
3.2.2 Free Private Key .....	16
3.2.3 Create Public Key .....	17
3.2.4 Free Public Key.....	17
<b>4 X.509 CERTIFICATE VALIDATION .....</b>	<b>18</b>
4.1 DATA TYPES.....	18
4.1.1 X.509 Certificate .....	18
4.1.2 X.509 Certificate Revocation List .....	18

4.1.3 X.509 Super-Type Object.....	18
4.1.4 Stack of X509_INFO.....	18
4.1.5 X.509 Certificate Chain .....	18
4.1.6 X.509 Name.....	18
4.1.7 X.509 Signature Algorithm .....	18
4.1.8 Certificate Verification Context Handle .....	19
4.1.9 Certificate Verification Return Code.....	19
4.2 GENERIC FUNCTIONS.....	20
4.2.1 Load X.509 Objects from PEM-format File.....	22
4.2.2 Load X.509 Certificate from a PEM-format File .....	23
4.2.3 Load X.509 Certificate from a Memory Buffer .....	24
4.2.4 Store X.509 Certificate into a Memory Buffer .....	25
4.2.5 Load X.509 CRL from a PEM-Format File.....	26
4.2.6 Load Private Key from a PEM-format File.....	27
4.2.7 Callback Function for PEM Format .....	28
4.2.8 Read X.509 Certificate Signature Algorithm .....	29
4.2.9 Read X.509 Certificate Names.....	30
4.2.10 X.509 Certificate Verification.....	32
4.2.11 Private Key Verification .....	35
4.2.12 Get Public Key from X.509 Certificate.....	36
4.2.13 Helper Function .....	37
<b>5 DIGEST .....</b>	<b>45</b>
5.1 DATA TYPES.....	45
5.1.1 SHA-256 Digest Buffer Length.....	45
5.2 GENERIC FUNCTIONS.....	46
5.2.1 X.509 Name SHA-256 Digest.....	47
5.2.2 SHA-256 Digest.....	48
<b>6 DIFFIE-HELLMAN KEY PAIR.....</b>	<b>49</b>
6.1 DATA TYPES.....	49
6.1.1 DH Key Pair.....	49
6.1.2 DH Public Key.....	49
6.1.3 DH Key Algorithm Type Constants .....	49
6.2 GENERIC FUNCTIONS.....	50
6.2.1 DH Key Pair Generation .....	51
6.2.2 Get DH Public Key Data Length.....	52

6.2.3 Get MODP DH Prime Data Length .....	53
6.2.4 Get MODP DH Generator Data Length.....	54
6.2.5 Get DH Public Key Data String .....	55
6.2.6 Get MODP DH Prime Data String.....	56
6.2.7 Get MODP DH Generator Data String.....	57
6.2.8 Helper Functions.....	58
<b>7 ASYMMETRIC DIGEST SIGNATURE AND DIGEST VERIFICATION .....</b>	<b>60</b>
7.1 DATA TYPES.....	60
7.1.1 Digest Signature Context Handle .....	60
7.2 GENERIC FUNCTIONS.....	61
7.2.1 Operations of Digest Signature Context Handle.....	62
7.2.2 Digest Signature Operation Initialization.....	64
7.2.3 Digest Signature Update.....	65
7.2.4 Get Size of Buffer for Digest Signature.....	66
7.2.5 Digest Signature Final .....	67
7.2.6 Digest Verification Operation Initialization.....	68
7.2.7 Digest Verification Update.....	69
7.2.8 Digest Verification Final .....	70
<b>8 SHAREDSECRET DERIVATION .....</b>	<b>71</b>
8.1 DATA TYPES.....	71
8.1.1 Derivation Context Handle .....	71
8.2 GENERIC FUNCTIONS.....	72
8.2.1 Operations of Derivation Context Handle.....	73
8.2.2 Derivation Initialization .....	74
8.2.3 Set Peer DH Public Key .....	75
8.2.4 Get the Size of SharedSecret .....	76
8.2.5 Derivation Execution .....	77
<b>9 HASH-BASED MESSAGE AUTHENTICATION CODE .....</b>	<b>78</b>
9.1 DATA TYPES.....	78
9.1.1 HMAC Key .....	78
9.1.2 HMAC Context Handle .....	78
9.2 GENERIC FUNCTIONS.....	79
9.2.1 Operations of HMAC Key .....	80
9.2.2 Operations of HMAC Context Handle.....	81
9.2.3 HMAC Operation Initialization.....	83

9.2.4 HMAC Update.....	84
9.2.5 HMAC Update.....	85
9.2.6 HMAC Compute Final.....	86
<b>10 AUTHENTICATED ENCRYPTION AND DECRYPTION.....</b>	<b>87</b>
10.1 DATA TYPES .....	87
10.1.1 AE Context Handle .....	87
10.1.2 AES Block Size .....	87
10.2 GENERIC FUNCTIONS .....	88
10.2.1 Operations of AE Context Handle.....	89
10.2.2 AE Operation Initialization.....	91
10.2.3 AE Additional Authenticated Data Update.....	93
10.2.4 AE Data Update.....	94
10.2.5 AE Encryption Final.....	96
10.2.6 AE Decryption Final.....	97
<b>11 RANDOM NUMBER GENERATION.....</b>	<b>98</b>
11.1 GENERIC FUNCTIONS .....	98
11.1.1 Random Number Generation.....	98
<b>12 LARGE INTEGERS .....</b>	<b>99</b>
12.1 DATA TYPES .....	99
12.1.1 BIGNUM .....	99
12.2 GENERIC FUNCTIONS .....	100
12.2.1 Large Integer Object Allocation.....	101
12.2.2 Large Integer Object Free.....	101
12.2.3 Count Number of Bytes in Large Integer .....	102
12.2.4 Convert Big Endian String to Large Integer.....	103
12.2.5 Convert Large Integer to Big Endian String.....	104
12.2.6 Generate Random Number in Large Integer .....	105

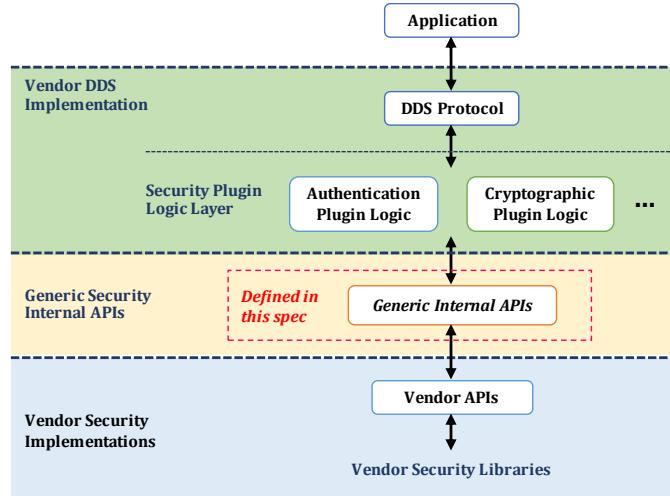
# 1 Overview

## 1.1 Introduction

This specification defines a set of APIs to provide DDS Security Plugins logic with a generic interface in Normal World, to the specific security libraries implementation, such as OpenSSL and arm DDS *SecureLib*. This set of APIs are called internally during DDS Security Plugins logic work flow.

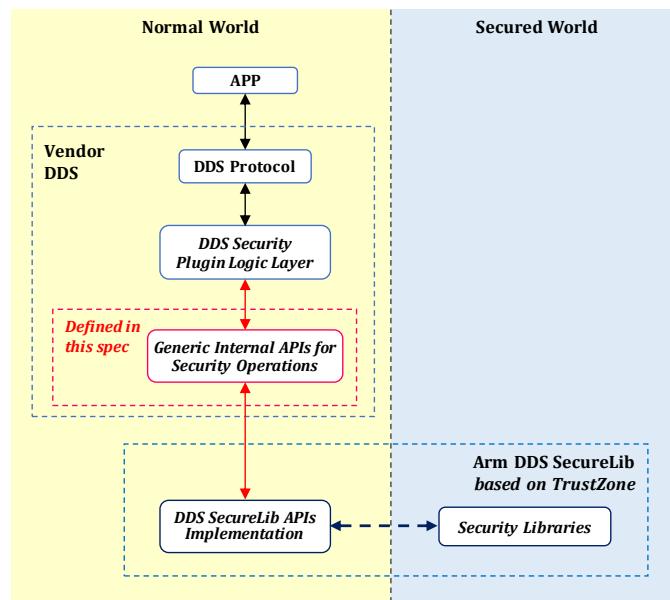
These internal APIs wrap the vendor security APIs which finally invoke the specific vendor security libraries.

The simplified hierarchy of DDS security with generic internal APIs is showed in Figure 1-1 below.



**FIGURE 1-1 GENERIC INTERNAL APIs INSIDE THE HIERARCHY OF DDS SECURITY**

An example of an implementation on arm platforms based on arm TrustZone is showed in Figure 1-2. In the implementation, generic APIs defined in this specification are implemented by Arm DDS *SecureLib* APIs. Arm DDS *SecureLib* APIs trigger the security libraries in Secure World based on arm TrustZone<sup>[1][2]</sup>. The security libraries complete the security operations requested in DDS Security Plugin logic and return the results.



**FIGURE 1-2 BLOCK DIAGRAM OF ARM DDS SECURITY SOLUTION**

This specification defines a general behavior for each API. Its specific implementation relies on the vendor security implementations.

The definitions of APIs in this specification should satisfy the requirements in DDS Security specification<sup>[5]</sup>.

## 1.2 References

- [1] Arm TrustZone: <https://www.arm.com/products/security-on-arm/trustzone>
- [2] Arm TrustZone on arm Developer: <https://developer.arm.com/technologies/trustzone>
- [3] DDS: Data-Distribution Service for Real-Time Systems version 1.4.  
<http://www.omg.org/spec/DDS/1.4/>
- [4] DDS-RTPS: Data-Distribution Service Interoperability Wire Protocol version 2.2.  
<http://www.omg.org/spec/DDSI-RTPS/2.2/>
- [5] DDS-SECURITY™: DDS Security Version 1.0. <http://www.omg.org/spec/DDS-SECURITY/1.0/>
- [6] GlobalPlatform Device Technology TEE System Architecture Version 1.1 Public Release.  
<https://www.globalplatform.org/specificationsdevice.asp>
- [7] GlobalPlatform Device Technology TEE Internal Core API Specification Version 1.1.1 Public Release. <https://www.globalplatform.org/specificationsdevice.asp>
- [8] GlobalPlatform Device Technology TEE Client API Specification Version 1.0 Public Release.  
<https://www.globalplatform.org/specificationsdevice.asp>
- [9] eProsima Fast RTPS: <http://www.eprosima.com/index.php/products-all/eprosima-fast-rtps>
- [10] eProsima Fast RTPS Documentation. <http://docs.eprosima.com/en/latest/>
- [11] eProsima Fast RTPS Security Documentation. <http://docs.eprosima.com/en/latest/security.html>
- [12] Security Services for DDS Security Plugins.

## 1.3 Revision History

Revision	Date	Description	Author
0.01	Nov 17 2017	Draft.	David Hu
0.1	Nov 27 2017	Update the document according to internal review.	David Hu
1.0	Nov 28 2017	The 1 <sup>st</sup> release version for external review.	David Hu

## 1.4 Terms and Definitions

### ***Data-Distribution Service for Real-Time Systems (DDS)***

The Object Management Group (OMG) open international middleware standard directly addressing publish-subscribe communications for real-time and embedded systems.

### ***Real-Time Publish-Subscribe DDS Interoperability Wire Protocol (DDS-I RTPS)***

A protocol standardized by OGM DDS for best effort and reliable pub-sub communications over unreliable transports in both unicast and multicast. **RTPS** will be used for short in this document.

#### ***Entity***

Base class for all RTPS entities. RTPS **Entity** represents the class of objects that are visible to other RTPS Entities on the network. As such, RTPS **Entity** objects have a globally unique identifier (GUID) and can be referenced inside RTPS messages.

#### ***Participant***

Container of all RTPS entities that share common properties and are located in a single address space.

For more details of DDS terms and definitions, please refer to OMG DDS Specification<sup>[3]</sup> and OMG RTPS specification<sup>[4]</sup>.

#### ***TrustZone***

Arm TrustZone technology is a System on Chip (SoC) and CPU system-wide approach to security. TrustZone is hardware-based security built into SoCs by semiconductor chip designers to provide a foundation for system-wide security and the creation of a trusted platform.

At the heart of the TrustZone approach is the concept of *Secure World* and *Normal World* that are hardware separated, with non-secure software blocked from accessing secure resources directly.

#### ***Secure World***

In arm TrustZone concepts, a Secure World is a physically isolated environment which provides confidentiality and integrity to the system.

It is used to protect high-value code and data for diverse use cases including authentication, payment, content protection and enterprise. Security assets can be protected in Secure World from software attacks and common hardware attacks. On application processors, it is frequently used to provide a security boundary for a GlobalPlatform Trusted Execution Environment (TEE).

#### ***Normal World***

In arm TrustZone concepts, rich Operating System and rich application environment run in a Normal World. The software running in the Normal Word cannot directly access the assets protected in Secure World.

For more details of arm TrustZone, please access the reference on arm websites<sup>[1][2]</sup>.

## 2 Generic Operations of Files and Formats

### 2.1 Data Types

#### 2.1.1 SEC\_IO

A SEC\_IO is an I/O stream abstraction of a stdio file or a memory buffer containing security information, such as a certificate or keys. A SEC\_IO represents the file or the chunk of data while operations are parsing or verifying the information stored in it.

The content of this structure is *implementation-defined*.

#### 2.1.2 ASN1\_OBJECT

An ASN1\_OBJECT represent a generic object in ASN1 format.

The content of this structure is *implementation-defined*.

#### 2.1.3 BUF\_MEM

A BUF\_MEM describes a chunk of data in a memory buffer.

The BUF\_MEM should at least include the fields which specify the data address and the length of the data. The required fields are listed below.

TABLE 2-1 REQUIRED FIELDS IN BUF\_MEM STRUCTURE

Field name	Descriptions
data	Address of the chunk of data
length	Length of data

Other fields in this structure are *implementation-defined*.

## 2.2 Generic Operations

Table 2-1 lists the operations to process generic file and format.

TABLE 2-2 OPERATIONS OF FILE AND FORMAT

Operations	Descriptions
<code>sec_file_new</code>	Create a new SEC_IO object to represent a stdio file.
<code>sec_mem_new</code>	Create a new SEC_IO object to represent a memory buffer.
<code>sec_mem_new_buf</code>	Create a new SEC_IO object by using a memory buffer which already allocated.
<code>sec_io_free</code>	Free a SEC_IO object
<code>sec_file_open_read</code>	Open a file for reading security information later.
<code>sec_mem_read_asn1</code>	Read ASN1 object and place it into a memory buffer in ASN1 format.
<code>set_sec_io_close</code>	set close status thus the underlying I/O stream should be closed when the SEC_IO object is freed.
<code>set_sec_io_noclose</code>	set close status thus the underlying I/O stream should not be closed when the SEC_IO object is freed.
<code>buf_mem_new</code>	Create a new BUF_MEM object.
<code>buf_mem_free</code>	Free a BUF_MEM object.
<code>get_sec_mem_ptr</code>	Place the underlying BUF_MEM object.

## 2.2.1 Create SEC\_IO for a File

```
SEC_IO* sec_file_new(void);
```

### Description

The function `sec_file_new` allocates, initializes and returns a `SEC_IO` object to represent a stdio file.

### Parameter

- No input parameter.

### Return

- `NULL` if an error occurs.
- The address of the `SEC_IO` object if the function succeeds.

## 2.2.2 Create SEC\_IO for a Memory Buffer

```
SEC_IO* sec_mem_new(void);
```

### Description

The function `sec_mem_new` allocates, initializes and returns a `SEC_IO` object to represent a memory buffer.

### Parameter

- No input parameter.

### Return

- `NULL` if an error occurs.
- The address of the `SEC_IO` object if the function succeeds.

## 2.2.3 Create SEC\_IO with an Existing Buffer

```
SEC_IO* sec_mem_new_buf(          const void*           buf,          int                 len);
```

### Description

The function `sec_mem_new_buf` creates a `SEC_IO` object using `len` bytes of data at `buf`.

### Parameter

- `buf`: data buffer address.
- `len`: length of data in `buf`.

### Return

- `NULL` if an error occurs.
- The address of the `SEC_IO` object if the function succeeds.

## 2.2.4 Free SEC\_IO

```
void sec_io_free(  
    SEC_IO* fp);
```

### Description

The function `sec_io_free` cleans up a `SEC_IO` object passed in `fp` and frees up the spaces allocated to it.

### Parameter

- `fp`: A pointer to the `SEC_IO` object. If it is `NULL`, the function will do nothing.

### Return

- No return

## 2.2.5 Open File for Reading

```
int sec_file_open_read(  
    SEC_IO*          fp,  
    char*            name);
```

### Description

The function `sec_file_open_read` set the `SEC_IO` object `fp` to use file `name` for reading.

If the target file is stored in Secure World, the open operation should protect the content in the target file.

### Parameter

- `fp`: A pointer to the `SEC_IO`.
- `name`: An identity of the file to be read.

### Return

- 0 if the operation fails for any reason.
- 1 if the operation succeeds.

## 2.2.6 Read ASN1 Object into SEC\_IO Memory Buffer

```
int sec_mem_read_asn1(
    SEC_IO*                      buf,
    const ASN1_OBJECT*           a);
```

### Description

The function `sec_mem_read_asn1` read an ASN1 object in `ASN1_OBJECT` and place it into a memory buffer `buf` in ASN1 format.

### Parameter

- `buf`: the memory buffer written with the ASN1 object.
- `a`: a buffer of ASN1 object.

### Return

- A negative value if the operation fails for a fatal error.
- The length of the actual data written into `buf` if the operation succeeds.

## 2.2.7 Set I/O Stream Close Status

### 2.2.7.1 Set I/O Stream as Close

```
void set_sec_io_close(
    SEC_IO* fp);
```

#### Description

The function `set_sec_io_close` set `fp` close status thus the underlying I/O stream should be closed when the `fp` is freed.

#### Parameter

- No parameter.

#### Return

- No return.

### 2.2.7.2 Set I/O Stream as Non-Close

```
void set_sec_io_noclose(
    SEC_IO* fp);
```

#### Description

The function `set_sec_io_noclose` set `fp` close status thus the underlying I/O stream should not be closed when the `fp` is freed.

#### Parameter

- No parameter.

#### Return

- No return.

## 2.2.8 BUF\_MEM Allocation

```
BUF_MEM* buf_mem_new(void);
```

### Description

The function `buf_mem_new` allocates, initializes and returns a `BUF_MEM` object.

### Parameter

- No input parameter.

### Return

- `NULL` if an error occurs.
- The address of the `BUF_MEM` object if the function succeeds.

## 2.2.9 BUF\_MEM Free

```
void buf_mem_free(  
    BUF_MEM* buf);
```

### Description

The function `buf_mem_free` cleans up a `BUF_MEM` object passed in `buf` and frees up the spaces allocated to it.

### Parameter

- `buf`: A pointer to the `BUF_MEM` object. If it is `NULL`, the function will do nothing.

### Return

- No return

## 2.2.10 Get the Pointer of BUF\_MEM

```
void get_sec_mem_ptr(  
    SEC_IO* buf,  
    BUF_MEM** pp);
```

### Description

The function `get_sec_mem_ptr` places the underlying `BUF_MEM` object indicating the `SEC_IO`, into `pp`.

### Parameter

- `buf`: a pointer to the `SEC_IO`.
- `pp`: stores the address of the underlying `BUF_MEM`.

### Return

- No return.

# 3 Generic Asymmetric Key Operation

The following sections specify the generic asymmetric keys. The structures defined in this chapter describe the generic model of asymmetric keys used in diverse algorithms.

## 3.1 Data Types

### 3.1.1 Private Key Handle

A `PRIV_KEY` is a reference to a private key.

If a private key is protected in Secure World based on arm TrustZone, `PRIV_KEY` should not contain the plaintext of the private key in Normal World.

The content of this handle is *implementation-defined*.

### 3.1.2 Public Key Handle

A `PUB_KEY` is a reference to a public key. It can contain the information of a public key extracted from a X.509 certificate. It can be used to decrypt a message encrypted by the corresponding private key or verify a signature signed by the corresponding private key.

The content of this handle is *implementation-defined*.

## 3.2 Generic Functions

Table 3-1 lists the operations required in asymmetric key operations.

TABLE 3-1 GENERIC SYMMETRIC KEY OPERATIONS LIST

Operations	Descriptions
<code>PRIV_KEY_new</code>	Allocate a new private key.
<code>PRIV_KEY_free</code>	Free a private key.

### 3.2.1 Create Private Key

```
PRIV_KEY* priv_key_new(void);
```

#### Description

The function `priv_key_new` allocates, initializes and returns an empty private key. The content of the private key object will be filled in specific key generation.

#### Parameter

- No input parameter.

#### Return

- `NULL` if an error occurs.
- The address of the new private key if the function succeeds.

### 3.2.2 Free Private Key

```
void priv_key_free(  
    PRIV_KEY* pkey);
```

#### Description

The function `priv_key_free` cleans up a private key passed in `pkey` and frees up the spaces allocated to it.

#### Parameter

- `pkey`: A pointer to the private key. If it is `NULL`, the function will do nothing.

#### Return

- No return

### 3.2.3 Create Public Key

```
PUB_KEY* pub_key_new(void);
```

#### Description

The function `pub_key_new` allocates, initializes and returns an empty public key. The content of the public key object will be filled in specific key generation.

#### Parameter

- No input parameter.

#### Return

- `NULL` if an error occurs.
- The address of the new public key if the function succeeds.

### 3.2.4 Free Public Key

```
void pub_key_free(  
    PUB_KEY* pubkey);
```

#### Description

The function `pub_key_free` cleans up a public key passed in `pubkey` and frees up the spaces allocated to it.

#### Parameter

- `pubkey`: A pointer to the public key. If it is `NULL`, the function will do nothing.

#### Return

- No return

## 4 X.509 Certificate Validation

The following sections specify the data types and functions for X.509 certification verification in DDS Security Authentication Plugin *DDS:Auth:PKI-DH*<sup>[5]</sup>.

### 4.1 Data Types

#### 4.1.1 X.509 Certificate

A `X509_CERT` object is a reference to X.509 certificate. The detailed definitions of `X509_CERT` is *implementation-defined*.

#### 4.1.2 X.509 Certificate Revocation List

A `X509_CRL` object represents a Certificate Revocation List (CRL). The detailed content of `X509_CRL` is *implementation-defined*.

#### 4.1.3 X.509 Super-Type Object

A `X509_INFO` is a super-type object, which can contain the information of a CRL or a certificate. The detailed definitions of `X509_INFO` is *implementation-defined*.

A `X509_INFO_STACK` consists of multiple `X509_INFOS`, to contain diverse objects extracted from certificate files. The detailed definitions are *implementation-defined*.

#### 4.1.4 Stack of `x509_INFO`

A `X509_INFO_STACK` contains references to multiple `X509_INFO` objects. `X509_INFO_STACK` should record the current number of `X509_INFO` included.

The detailed definitions are *implementation-defined*.

#### 4.1.5 X.509 Certificate Chain

A `X509_CHAIN` object contains a X.509 certificate chain, including the necessary objects and information for a X.509 certificate verification. The detailed definitions are *implementation-defined*.

#### 4.1.6 X.509 Name

A `X509_NAME` object represents the name of a field in a X.509 certificate. The detailed definitions are *implementation-defined*.

#### 4.1.7 X.509 Signature Algorithm

A `X509_ALG` object indicates the signature algorithm in a X.509 certificate.

The `X509_ALG` should contain the algorithm described in ASN1 format. The required field is listed below.

**TABLE 4-1 REQUIRED FIELDS IN `x509_ALGOR` STRUCTURE**

Field name	Descriptions
<code>algorithm</code>	Signature algorithm in ASN1 format.

Other fields in this structure are *implementation-defined*.

#### 4.1.8 Certificate Verification Context Handle

A `X509_VERF_CTX` is a handle of a certification verification operation. It should contain all the necessary information and objects required in the operation, including certificate chain and CRL, etc.

It should also record the last certificate verification return code. The return code is described in section 4.1.9.

The objects organized in `X509_VERF_CTX` is entirely *implementation-defined*.

#### 4.1.9 Certificate Verification Return Code

The return code indicates the results of X.509 certificate verification operation. All the return codes should be defined as type `uint32_t`.

The flag constant of verification success is showed in Table 4-2.

**TABLE 4-2 RETURN CODE OF CERTIFICATE VERIFICATION SUCCESS**

Name	Value	Comment
<code>X509_V_OK</code>	0x00000000	Return code to indicate that a certificate is successfully verified.

All the error codes are *implementation-defined*.

## 4.2 Generic Functions

Table 4-3 lists the operations required in X.509 certificate.

TABLE 4-3 X.509 OPERATIONS LIST

Operations	Descriptions
<code>x509_read_file_pem_info</code>	Load X.509 objects from a PEM-format file.
<code>x509_read_file_pem_cert</code>	Load a X.509 certificate from a PEM-format file.
<code>x509_read_mem_pem_cert</code>	Load a X.509 certificate from PEM-format data in a memory buffer
<code>x509_write_mem_pem_cert</code>	Write a X.509 certificate into a memory buffer in PEM format.
<code>x509_read_file_pem_crl</code>	Load a X.509 Certificate Revocation List (CRL) from a PEM-format file.
<code>pem_file_read_private_key</code>	Load a private key from a PEM-format file
<code>pem_password_cb</code>	Callback function to encrypt/decrypt the data written into or read from PEM-format files or data
<code>x509_get_sign_alg</code>	Get the signature algorithm of a X.509 certificate.
<code>x509_get_subject_name</code>	Get the Subject Name of a X.509 certificate.
<code>x509_name_get_str</code>	Get the ASCII version of a name in a X.509 certificate.
<code>x509_name_free_str</code>	Free the ASCII string got from <code>x509_name_get_str</code>
<code>x509_cerf_verf_ctx_new</code>	Allocate X.509 certificate verification context handle.
<code>x509_cerf_verf_ctx_free</code>	Free X.509 certificate verification context handle.
<code>x509_cert_verf_init</code>	Initialize the verification of a X.509 certificate
<code>x509_cert_verf</code>	Verify a X.509 certificate
<code>x509_cert_verf_cleanup</code>	Clean up the verification of a X.509 certificate
<code>x509_verf_priv_key</code>	Verify a private key associated with a X.509 certificate
<code>x509_get_pub_key</code>	Get a public key from a X.509 certificate
<code>x509_chain_new</code>	Allocate a <code>X509_CHAIN</code>
<code>x509_chain_free</code>	Free the space allocated to a <code>X509_CHAIN</code>
<code>x509_info_new</code>	Allocate a <code>X509_INFO</code>
<code>x509_info_free</code>	Free the space allocated to a <code>X509_INFO</code>
<code>stack_x509_info_new</code>	Allocate an empty <code>STACK_X509_INFO</code>
<code>stack_x509_info_free</code>	Free a <code>STACK_X509_INFO</code> structure
<code>stack_x509_info_push</code>	Add a <code>X509_INFO</code> on the tail of <code>STACK_X509_INFO</code>

<b>stack_x509_info_pop</b>	Remove the last X509_INFO in STACK_X509_INFO
<b>stack_x509_info_pop_free</b>	Clean up and free STACK_X509_INFO and its all X509_INFO included.
<b>stack_x509_info_num</b>	Get the number of X509_INFO in STACK_X509_INFO.
<b>stack_x509_info_pick</b>	Get a X509_INFO from STACK_X509_INFO according to the index.
<b>x509_chain_add_cert</b>	Add a X.509 certificate into a X509_CHAIN
<b>x509_chain_add_crl</b>	Add a X.509 CRL into a X509_CHAIN

## 4.2.1 Load X.509 Objects from PEM-format File

```
x509_info_stack* x509_read_file_pem_info(  
    SEC_IO*           fp,  
    pem_password_cb* cb,  
    void*             u);
```

### Description

The function `x509_read_file_pem_info` reads X.509 objects, such as a X.509 CA or a X.509 CRL, identified by the PEM-format file passed in `fp`. It can parse X.509 objects and extract out items which can be exposed in Normal World. Any field which should be protected in Secure World must not be accessed in Normal World nor be stored into `x509_INFO_STACK`.

The items extracted out should be stored into `x509_INFO_STACK`, which is allocated inside `x509_read_file_pem_info`.

The specific operation can be executed in Normal World or Secure World. It depends on the implementation and security requirement.

### Parameter

- `fp`: `SEC_IO` which contains an identity of the X.509 objects. If it is `NULL`, the behavior of the function `x509_read_file_pem_info` is *implementation-defined*. It can either return error or load a default object if it exists.
- `pem_password_cb`: Callback function to decrypt the objects.
- `u`: password to decrypt the objects.

### Return

- `NULL` if an error occurs.
- The address of `x509_INFO_STACK` if the operation succeeds.

## 4.2.2 Load X.509 Certificate from a PEM-format File

```
X509_CERT* x509_read_file_pem_cert(  
    SEC_IO*          fp,  
    pem_password_cb* cb,  
    void*            u);
```

### Description

The function `x509_read_file_pem_cert` reads a X.509 certificate signed by the root CA in PEM format containing the signed public key. The PEM format file is passed in `file`.

The function `x509_read_file_pem_cert` parses the certificate and stores the fields into `X509_CERT`, which is allocated inside `x509_read_file_pem_cert`. Any information which should be protected in Secure World should neither be accessed in Normal World nor be stored into `X509_CERT`.

The specific operation can be executed in Normal World or Secure World. It depends on the implementation and security requirement.

### Parameter

- `fp`: `SEC_IO` which contains an identity of the X.509 certificate. If it is `NULL`, the behavior of the function `x509_read_file_pem_cert` is *implementation-defined*. It can either return error or load a default certificate if it exists.
- `pem_password_cb`: Callback function to decrypt the certificate.
- `u`: password to decrypt the certificate.

### Return

- `NULL` if an error occurs.
- The address of `X509_CERT` if the operation succeeds.

### 4.2.3 Load X.509 Certificate from a Memory Buffer

```
X509_CERT* x500_read_mem_pem_cert(  
    SEC_IO*           buf,  
    pem_password_cb* cb,  
    void*            u);
```

#### Description

The function `x500_read_mem_pem_cert` reads a X.509 certificate signed by the root CA in PEM format containing the signed public key. The certificate in PEM format is stored in a memory buffer passed in `buf`.

The function `x500_read_mem_pem_cert` parses the certificate and stores the fields into `X509_CERT`, which is allocated inside `x500_read_mem_pem_cert`. Any information which should be protected in Secure World should neither be accessed in Normal World nor be stored into `X509_CERT`.

The specific operation can be executed in Normal World or Secure World. It depends on the implementation and security requirement.

#### Parameter

- `buf`: `SEC_IO` which contains an identity of the X.509 certificate. If it is `NULL`, the behavior of the function `x500_read_mem_pem_cert` is *implementation-defined*. It can either return error or load a default certificate if it exists.
- `pem_password_cb`: Callback function to decrypt the certificate.
- `u`: password to decrypt the certificate.

#### Return

- `NULL` if an error occurs.
- The address of `X509_CERT` if the operation succeeds.

#### 4.2.4 Store X.509 Certificate into a Memory Buffer

```
int x509_write_mem_pem_cert(  
    SEC_IO*           buf,  
    X509_CERT*        cert);
```

##### Description

The function `x509_write_mem_pem_cert` writes a X.509 certificate in PEM format into a memory buffer passed in `buf`.

The specific operation can be executed in Normal World or Secure World. It depends on the implementation and security requirement.

##### Parameter

- `buf`: `SEC_IO` written with the certificate.
- `pem_password_cb`: Callback function to encrypt the certificate.
- `u`: password to encrypt the certificate.

##### Return

- 0 if the write fails for any reason.
- 1 if the write succeeds.

## 4.2.5 Load X.509 CRL from a PEM-Format File

```
x509_crl* x509_read_file_pem_crl(  
    SEC_IO*          fp,  
    pem_password_cb* cb,  
    void*            u);
```

### Description

The function `x509_read_file_pem_crl` reads a X.509 CRL from the file in PEM format passed in `file`.

The function `x509_read_file_pem_crl` parses the CRL and stores the fields into `x509_CRL`, which is allocated inside `x509_read_file_pem_crl`.

The specific operation can be executed in Normal World or Secure World. It depends on the implementation and security requirement.

### Parameter

- `fp`: `SEC_IO` which contains an identity of the X.509 CRL. If it is `NULL`, the behavior of the function `x509_read_file_pem_crl` is *implementation-defined*. It can either return error or load a default certificate if it exists.
- `pem_password_cb`: Callback function to decrypt the CRL.
- `u`: password to decrypt the CRL.

### Return

- `NULL` if an error occurs.
- The address of `x509_CRL` if the operation succeeds.

## 4.2.6 Load Private Key from a PEM-format File

```
PRIV_KEY* pem_file_read_private_key (
    SEC_IO*           fp,
    pem_password_cb* cb,
    void*             u);
```

### Description

The function `pem_file_read_private_key` reads a private key from the file in PEM format passed in `file`.

The specific operation can be executed in Normal World or Secure World. It depends on the implementation and security requirement.

If the private key is protected in Secure World based on arm TrustZone, the specific operation should be executed in Secure World. Furthermore, the operation should avoid leaking the root private key.

### Parameter

- `fp`: `SEC_IO` which contains an identity of the private key. If it is `NULL`, the behavior of the function `pem_file_read_private_key` is *implementation-defined*. It can either return error or load a default certificate if it exists.
- `pem_password_cb`: Callback function to decrypt the private key.
- `u`: password to decrypt the private key.

### Return

- 0 if the read fails for any reason.
- 1 if the read succeeds.

## 4.2.7 Callback Function for PEM Format

```
int pem_password_cb(  
    char*          buf,  
    int            size,  
    int            rwflag,  
    void*          u);
```

### Description

When installing security information, a password might be used to encrypt the information for additional protection.

The password callback `pem_password_cb` hands back the password to be used during decryption. On invocation, a pointer to `u` is provided. The password callback must write the password into the provided buffer `buf` which is of size `size`. The actual length of the password must be returned to the calling function. `rwflag` indicates whether the callback is used for reading/decryption (`rwflag = 0`) or writing/encryption (`rwflag = 1`).

### Parameter

- `buf`: a reference to the buffer written with the password hung back.
- `size`: the length of `buf`.
- `rwflag`: indicates whether the callback is used for reading/decryption (`rwflag = 0`) or writing/encryption (`rwflag = 1`).
- `u`: the password.

### Return

- The actual length of the password.

## 4.2.8 Read X.509 Certificate Signature Algorithm

```
void x509_get_sign_alg(  
    X509_ALGOR** sign_alg);  
    X509_CERT* cert,
```

### Description

The function `x509_get_sign_alg` gets the signature algorithm used in the X.509 certificate passed in `cert`.

The address of the signature algorithm string should be set into `sign_alg`.

The specific operation can be executed in Normal World or Secure World. It depends on the implementation and security requirement.

### Parameter

- `sign_alg`: A pointer to the signature algorithm string should be placed in this pointer. If the function fails for any reason, the value in `sign_alg` should be treated as invalid.
- `cert`: A pointer to the X.509 certificate. It must not be `NULL`.

### Return

- No return.

## 4.2.9 Read X.509 Certificate Names

### 4.2.9.1 Get X.509 Certificate Subject Name

```
X509_NAME* x509_get_subject_name(  
    X509_CERT* cert);
```

#### Description

The function `x509_get_subject_name` gets the subject name in the X.509 certificate passed in `cert`.

The specific operation can be executed in Normal World or Secure World. It depends on the implementation and security requirement.

#### Parameter

- `cert`: A pointer to the X.509 certificate. It must not be `NULL`.

#### Return

- The reference to the subject name if the function succeeds.
- `NULL` if the function fails for any reason.

### 4.2.9.2 Get X.509 Certificate Name ASCII String

```
char* x509_name_get_str(  
    X509_NAME* name);
```

#### Description

The function `x509_name_get_str` gets the ASCII string of a X.509 name passed in `X509_NAME`.

The function `x509_name_free_str` should be called after function `x509_name_get_str`. See section 4.2.9.3 for details of function `x509_name_free_str`.

The specific operation can be executed in Normal World or Secure World. It depends on the implementation and security requirement.

#### Parameter

- `X509_NAME`: A reference to the X.509 name. It must not be `NULL`.

#### Return

- The reference to the string if the function succeeds.
- `NULL` if the function fails for any reason.

### 4.2.9.3 Free X.509 Certificate Subject Name String

```
void x509_name_free_str(  
    char* str);
```

#### Description

The function `x509_name_free_str` free the string passed in `str`. The `str` is returned from function `x509_name_get_str` previously called.

The function `x509_name_free_str` should be called after function `x509_name_get_str`. See section 4.2.9.2 for details of the function `x509_name_get_str`.

The specific operation can be executed in Normal World or Secure World. It depends on the implementation and security requirement.

#### **Parameter**

- `str`: The string to be freed. If it is `NULL`, the function will do nothing.

#### **Return**

- No return.

## 4.2.10 X.509 Certificate Verification

### 4.2.10.1 Certificate Verification Context Allocation

```
X509_VERF_CTX* x509_cert_verf_ctx_new(void);
```

#### Description

The function `x509_verf_ctx_new` allocates a new context handle for certificate verification.

The specific operation can be executed in Normal World. It depends on the implementation and security requirement.

#### Parameter

- No input parameter.

#### Return

- `NULL` if the function fails for any reason.
- The pointer to `X509_VERF_CTX`.

### 4.2.10.2 Certificate Verification Context Free

```
void x509_cert_verf_ctx_free(  
    X509_VERF_CTX*      ctx);
```

#### Description

The function `x509_verf_ctx_free` frees and cleans up a `X509_VERF_CTX` object passed in `ctx`.

The specific operation can be executed in Normal World. It depends on the implementation and security requirement.

#### Parameter

- `ctx`: A pointer to the `X509_VERF_CTX` object to be freed. If it is `NULL`, the function will do nothing.

#### Return

- No return.

### 4.2.10.3 Certificate Verification Operation Initialization

```
int x509_cert_verf_init(  
    X509_VERF_CTX*      ctx,  
    X509_CHAIN*          chain,  
    X509_CERT*           cert,  
    bool                 is_crl);
```

#### Description

The function `x509_cert_verf_init` initializes the context of a certificate verification. The initialization process can include setting flags.

The function `x509_cert_verf_init` should be called prior to function `x509_cert_verf`.

The specific operation can be executed in Normal World or Secure World. It depends on the implementation and security requirement.

#### Parameter

- `ctx`: A pointer to the context handle on be initialized for following certificate verification. It must not be `NULL`.
- `chain`: A pointer to the X.509 certificate chain for verification.
- `cert`: A pointer to the X.509 identity certificate for verification.
- `is_crl`: A boolean flag to indicate if a CRL exists.

#### Return

- 1 if the initialization succeeds.
- 0 if the initialization fails.

### 4.2.10.4 Certificate Verification

```
int x509_cert_verf(  
    X509_VERF_CTX*      ctx);
```

#### Description

The function `x509_cert_verf` execute the certificate verification.

The function `x509_cert_verf` should be called only after `ctx` is initialized in function `x509_cert_verf_init`.

The function `x509_cert_verf` should be followed by function `x509_cert_verf_cleanup`.

The specific operation can be executed in Normal World or Secure World. It depends on the implementation and security requirement.

#### Parameter

- `ctx`: A pointer to the context handle for certificate verification. The context handle has already been initialized in function `x509_cert_verf_init`. It must not be `NULL`.

#### Return

- 1 if the verification succeeds.
- 0 if the verification fails.

### 4.2.10.5 Certificate Verification

```
int x509_cert_verf_get_err(  
    X509_VERF_CTX*      ctx);
```

#### Description

The function `x509_cert_verf_get_err` returns the error code of `ctx`.

The function `x509_cert_verf_get_err` should be called only after `x509_cert_verf` completes.

The specific operation can be executed in Normal World or Secure World. It depends on the implementation and security requirement.

#### Parameter

- `ctx`: A pointer to the context handle for certificate verification. The context handle has already been initialized in function `x509_cert_verf_init`. It must not be `NULL`.

#### Return

- Return code: See section 4.1.9 for the details of return code.

### 4.2.10.6 Certificate Verification Operation Cleanup

```
void x509_cert_verf_cleanup(  
    X509_VERF_CTX*           ctx);
```

#### Description

The function `x509_cert_verf_cleanup` cleans up the context of the certificate verification. The function `x509_cert_verf_cleanup` should be called after function `x509_cert_verf`. The specific operation can be executed in Normal World or Secure World. It depends on the implementation and security requirement.

#### Parameter

- `ctx`: A pointer to the context handle for previous certificate verification. The corresponding context will be cleaned up. It must not be `NULL`.

#### Return

- No return.

## 4.2.11 Private Key Verification

```
int x509_verf_private_key(  
    X509_CERT* cert,  
    PRIV_KEY* pkey);
```

### Description

The function `x509_verf_private_key` verifies a private key associated with a X.509 certificate. If the function succeeds, a handle on the private key will be returned.

If the private key is protected in Secure World based on arm TrustZone, the specific operation should be executed in Secure World. Furthermore, the handle should avoid containing any information which might leak private key.

### Parameter

- `cert`: A pointer to the identity certificate which is associated to the private key. It must not be `NULL`.
- `pkey`: An identity of the private key. If it is `NULL`, the behavior of the function is *implementation-defined*.

### Return

- 1 if the verification succeeds.
- 0 if the verification fails.

## 4.2.12 Get Public Key from X.509 Certificate

```
PUB_KEY* x509_get_pub_key(  
    X509_CERT* cert);
```

### Description

The function `x509_get_pub_key` extracts the public key in a X.509 certificate passed in `cert`. If the function succeeds, a handle on the public key will be returned.

### Parameter

- `cert`: A pointer to the identity certificate which contains the target public key. It must not be `NULL`.

### Return

- `NULL` if the function fails for any reason.
- The pointer to `PUB_KEY` of the public key.

## 4.2.13 Helper Function

The following helper functions deal with the data types defined in section 4.1. The helper functions are usually *implementation-defined*.

### 4.2.13.1 X.509 Certificate Object Allocation

```
X509_CERT* x509_cert_new(void);
```

#### Description

The function `x509_cert_new` allocate an empty `X509_CERT` structure.

The specific operation can be executed in Normal World. It depends on the implementation and security requirement.

#### Parameter

- No input parameter.

#### Return

- `NULL` if the function fails for any reason.
- The pointer to `X509_CERT`.

### 4.2.13.2 X.509 Certificate Object Free

```
void x509_cert_free(
    X509_CERT* cert);
```

#### Description

The function `x509_cert_free` frees and cleans up a `X509_CERT` object passed in `cert`.

The specific operation can be executed in Normal World. It depends on the implementation and security requirement.

#### Parameter

- `cert`: A pointer to the `X509_CERT` object to be freed. If it is `NULL`, the function will do nothing.

#### Return

- No return.

### 4.2.13.3 X.509 Certificate Chain Allocation

```
x509_chain* x509_chain_new(void);
```

#### Description

The function `x509_chain_new` allocate an empty X.509 certificate chain.

The specific operation can be executed in Normal World. It depends on the implementation and security requirement.

#### Parameter

- No input parameter.

#### Return

- NULL if the function fails for any reason.
- The pointer to `X509_CHAIN`.

### 4.2.13.4 X.509 Certificate Chain Free

```
void x509_chain_free(  
                      X509_CHAIN*      chain);
```

#### Description

The function `x509_chain_free` frees and cleans up a `X509_CHAIN` object passed in `chain`.

The specific operation can be executed in Normal World. It depends on the implementation and security requirement.

#### Parameter

- `chain`: A pointer to the `X509_CHAIN` object to be freed. If it is NULL, the function will do nothing.

#### Return

- No return.

#### 4.2.13.5 X.509 CRL Allocation

```
x509_CRL* x509_crl_new(void);
```

##### Description

The function `x509_crl_new` allocate an empty X.509 CRL.

The specific operation can be executed in Normal World. It depends on the implementation and security requirement.

##### Parameter

- No input parameter.

##### Return

- `NULL` if the function fails for any reason.
- The pointer to `X509_CRL`.

#### 4.2.13.6 X.509 CRL Free

```
void x509_crl_free(  
                    X509_CRL*      crl);
```

##### Description

The function `x509_crl_free` frees and cleans up a `X509_CRL` object passed in `crl`.

The specific operation can be executed in Normal World. It depends on the implementation and security requirement.

##### Parameter

- `crl`: A pointer to the `X509_CRL` object to be freed. If it is `NULL`, the function will do nothing.

##### Return

- No return.

#### **4.2.13.7 x509\_INFO Allocation**

```
x509_INFO* x509_info_new(void);
```

##### **Description**

The function `x509_info_new` allocate an empty `x509_INFO` type object.

The specific operation can be executed in Normal World. It depends on the implementation and security requirement.

##### **Parameter**

- No input parameter.

##### **Return**

- `NULL` if the allocation fails for any reason.
- The pointer to `x509_INFO`.

#### **4.2.13.8 x509\_INFO Object Free**

```
void x509_info_free(  
                      X509_INFO* info);
```

##### **Description**

The function `x509_info_free` frees and cleans up a `X509_INFO` object passed in `info`.

The specific operation can be executed in Normal World. It depends on the implementation and security requirement.

##### **Parameter**

- `info`: A pointer to the `X509_INFO` object to be freed. If it is `NULL`, the function will do nothing.

##### **Return**

- No return.

#### 4.2.13.9 STACK\_X509\_INFO Allocation

```
STACK_X509_INFO* stack_x509_info_new(void);
```

##### Description

The function `stack_x509_info_new` allocate an empty `STACK_X509_INFO` type object.

The specific operation can be executed in Normal World. It depends on the implementation and security requirement.

##### Parameter

- No input parameter.

##### Return

- `NULL` if the allocation fails for any reason.
- The pointer to `STACK_X509_INFO`.

#### 4.2.13.10 STACK\_X509\_INFO Free

```
void stack_x509_info_free(  
    STACK_X509_INFO* sk);
```

##### Description

The function `stack_x509_info_free` frees a `STACK_X509_INFO` object passed in `sk`. The function `stack_x509_info_free` should not touch the `x509_INFO` included in the `STACK_X509_INFO`.

The specific operation can be executed in Normal World. It depends on the implementation and security requirement.

##### Parameter

- `sk`: A pointer to the `STACK_X509_INFO` object to be freed. If it is `NULL`, the function will do nothing.

##### Return

- No return.

#### 4.2.13.11 Add a x509\_INFO into STACK\_X509\_INFO

```
int stack_x509_info_push(  
    STACK_X509_INFO* sk,  
    X509_INFO* info);
```

##### Description

The function `stack_x509_info_push` add the reference to the `X509_INFO` object in `info` in the tail of the `STACK_X509_INFO` in `sk`.

The specific operation can be executed in Normal World. It depends on the implementation and security requirement.

##### Parameter

- `sk`: A pointer to the `STACK_X509_INFO`.
- `info`: A reference to the target `X509_INFO`.

#### Return

- 0 if the operation fails for any reason.
- 1 if the operation succeeds.

### 4.2.13.12 Remove a `x509_INFO` from `STACK_X509_INFO`

```
STACK_INFO* stack_x509_info_pop(
    STACK_X509_INFO* sk);
```

#### Description

The function `stack_x509_info_pop` remove the reference to the last `X509_INFO` object from a `STACK_X509_INFO` in `sk`.

The function `stack_x509_info_pop` should not free or modify the removed `X509_INFO` object.

The specific operation can be executed in Normal World. It depends on the implementation and security requirement.

#### Parameter

- `sk`: A pointer to the `STACK_X509_INFO` containing the target `X509_INFO`.

#### Return

- `NULL` if the function fails for any reason.
- The pointer to the removed `X509_INFO`.

### 4.2.13.13 `STACK_X509_INFO` Cleanup and Free

```
void stack_x509_info_pop_free(
    STACK_X509_INFO* sk);
```

#### Description

The function `stack_x509_info_pop_free` frees all the `X509_INFO` objects included in a `STACK_X509_INFO` in `sk`. After all the `X509_INFO` objects are cleaned up, the `STACK_X509_INFO` itself is finally freed.

The specific operation can be executed in Normal World. It depends on the implementation and security requirement.

#### Parameter

- `sk`: A pointer to the `STACK_X509_INFO` to be cleaned up and freed.

#### Return

- No return.

### 4.2.13.14 Get the Number of `x509_INFO` in `STACK_X509_INFO`

```
int stack_x509_info_num(
```

```
STACK_X509_INFO*      sk);
```

## Description

The function `stack_x509_info_num` returns the number of `X509_INFO` included inside `STACK_X509_INFO` passed in `sk`.

The specific operation can be executed in Normal World. It depends on the implementation and security requirement.

## Parameter

- `sk`: A pointer to the `STACK_X509_INFO` containing `X509_INFO`.

## Return

- The number of `X509_INFO` included.

### 4.2.13.15 Get a `x509_info` Object from `STACK_X509_INFO`

```
X509_INFO *stack_x509_info_pick(
    STACK_X509_INFO*      sk,
    int                   idx);
```

## Description

The function `stack_x509_info_pick` returns the reference to the `X509_INFO` object picked by index `idx` from a `STACK_X509_INFO` in `sk`.

The specific operation can be executed in Normal World. It depends on the implementation and security requirement.

## Parameter

- `sk`: A pointer to the `STACK_X509_INFO` containing the target `X509_INFO`.
- `idx`: the index of target `X509_INFO` in `STACK_X509_INFO`.

## Return

- `NULL` if the function fails for any reason.
- The pointer to the target `X509_INFO`.

#### 4.2.13.16 Add X.509 Certificate into Certificate Chain

```
void x509_chain_add_cert(  
    X509_CHAIN*      chain,  
    X509_CERT*       cert);
```

##### Description

The function `x509_chain_add_cert` adds a X.509 certificate passed in `cert` into the X.509 certificate chain passed in `chain`.

The specific operation can be executed in Normal World. It depends on the implementation and security requirement.

##### Parameter

- `chain`: A pointer to the X.509 certificate chain. It must not be NULL.
- `cert`: A pointer to the X.509 identity certificate. It must not be NULL.

##### Return

- No return

#### 4.2.13.17 Add X.509 CRL into Certificate Chain

```
void x509_chain_add_crl(  
    X509_CHAIN*      chain,  
    X509_CRL*        crl);
```

##### Description

The function `x509_chain_add_crl` adds a X.509 CRL in `crl` into the X.509 certificate chain passed in `chain`.

The specific operation can be executed in Normal World. It depends on the implementation and security requirement.

##### Parameter

- `chain`: A pointer to the X.509 certificate chain. It must not be NULL.
- `crl`: A pointer to the X.509 CRL. It must not be NULL.

##### Return

- No return

# 5 Digest

The following sections defines the data types and functions which support hash operations in DDS Security.

## 5.1 Data Types

### 5.1.1 SHA-256 Digest Buffer Length

SHA256\_DIGEST\_LENGTH defines the data buffer length to hold the SHA-256 digest result.

**TABLE 5-1 CONSTANT OF SHA-256 DIGEST BUFFER LENGTH**

Constant Name	Value
SHA256_DIGEST_LENGTH	32

## 5.2 Generic Functions

Digest interface should provide the APIs shown in Table 5-2.

TABLE 5-2 DIGEST OPERATIONS

Operations	Descriptions
<b>x509_name_sha256</b>	Hash a name in a X.509 certificate with SHA-256
<b>sha256</b>	Hash data with SHA-256.

## 5.2.1 X.509 Name SHA-256 Digest

```
int x509_name_sha256(
    X509_NAME*           name,
    unsigned char*        buf,
    unsigned int          length);
```

### Description

The function `x509_name_sha256` hashes a X.509 certificate name passed in `name`. The hashed result is put into `buf`.

SHA-256 is in use. The detailed implementation should satisfy the requirement in OMG DDS Security specification.

### Parameter

- `name`: A pointer to the X.509 certificate name to be hashed. It must not be `NULL`.
- `buf`: A pointer to the buffer to contain the hashed data. It must not be `NULL`. Furthermore, the size of the buffer should be large enough to hold the hashed data.
- `length`: length of the hashed data.

### Return

- 0 if the hash fails for any reason.
- 1 if the hash succeeds.

## 5.2.2 SHA-256 Digest

```
int sha256(  
    const void*      data,  
    size_t           count,  
    unsigned char*   buf);
```

### Description

The function `sha256` hashes the data passed in `data` with size in `count`. The hashed result is put into `buf`.

SHA-256 is in use. The detailed implementation should satisfy the requirement in OMG DDS Security specification.

### Parameter

- `data`: A pointer to the data buffer to be hashed. It must not be `NULL`.
- `count`: The length of `data`.
- `buf`: Reference to the buffer to contain the hashed data. It must not be `NULL`. Furthermore, the size of the buffer should be large enough to hold the hashed data.

### Return

- `0` if the hash fails for any reason.
- `1` if the hash succeeds.

# 6 Diffie-Hellman Key Pair

The following sections describe the data types and functions for Diffie-Hellman (DH) key pair generation. The DH key pair is used in DH message handshake in DDS Security Authentication Plugin *DDS:Auth:PKI-DH*<sup>[5]</sup>.

## 6.1 Data Types

### 6.1.1 DH Key Pair

A `DH_KEY` object represents a DH key pair. A `DH_KEY` should contain the pair of a private DH key and the corresponding public one.

A `DH_KEY` should contain a field which specifies its DH algorithm type.

The detailed definitions of `DH_KEY` is *implementation-defined*.

### 6.1.2 DH Public Key

A `DH_PUB_KEY` structure represents a DH public key, especially the public key information received from the peer party. It should contain the DH key pair parameters shared in DH exchange.

A `DH_PUB_KEY` should contain a field which specifies its DH algorithm type.

The implementation of `DH_PUB_KEY` should follow DDS Security specification. The details of `DH_PUB_KEY` is *implementation-defined*.

### 6.1.3 DH Key Algorithm Type Constants

The DH key algorithm type constants indicate the algorithm type of DH key to be generated. All the constants should be defined as DDS Security specification requires.

The constants are showed in Table 6-1.

TABLE 6-1 DH KEY ALGORITHM TYPES

Name	Key Agreement Algorithm
<code>TYPE_DH_KEYPAIR</code>	“DH+MODP-2048-256”.
<code>TYPE_ECDH_KEYPAIR</code>	“ECDH+prime256v1-CEUM”

The specific value of each type depends on vendor implementation.

All the other type definitions are invalid.

## 6.2 Generic Functions

Table 6-2 list the APIs for DH key operations.

TABLE 6-2 DH KEY OPERATIONS

Operations	Descriptions
<code>dh_gen</code>	Generate a pair of DH key.
<code>dh_key_new</code>	Create an empty DH key pair structure
<code>dh_key_free</code>	Free and clean up a DH key pair object.
<code>dh_pub_key_new</code>	Create an empty DH public key structure.
<code>dh_pub_key_free</code>	Free and clean up a DH public key object.
<code>dh_get_pub_key_len</code>	Get the length of DH public key data.
<code>dh_get_p_len</code>	Get the length of MODP DH prime ( <i>p</i> ) data.
<code>dh_get_g_len</code>	Get the length of MODP DH generator ( <i>g</i> ) data.
<code>dh_get_pub_key_data</code>	Get the octet string of DH public key data in MSB format.
<code>dh_get_p_data</code>	Get the octet string of MODP DH prime ( <i>p</i> ) data in MSB format.
<code>dh_get_g_data</code>	Get the octet string of MODP DH generator ( <i>g</i> ) data in MSB format.

## 6.2.1 DH Key Pair Generation

```
DH_KEY* dh_gen(  
    int             type);
```

### Description

The function `dh_gen` generates a pair of DH key according to the algorithm specified by `type`.

### Parameter

- `type`: The type of DH key algorithm type. The valid types are showed in Table 6-1.

### Return

- `NULL` if the generation fails for any reason.
- The pointer to a `DH_KEY` object generated if the function succeeds.

## 6.2.2 Get DH Public Key Data Length

```
int dh_get_pub_key_len(  
    DH_KEY*           dh);
```

### Description

The function `dh_get_pub_key_len` returns the length of the binary representation of DH public key data.

### Parameter

- `dh`: The pointer to the DH key pair `DH_KEY` which contains the public key.

### Return

- 0 or a negative value if the generation fails for any reason.
- The length of the binary representation of DH public key if the function succeeds.

### 6.2.3 Get MODP DH Prime Data Length

```
int dh_get_p_len(  
    DH_KEY* dh);
```

#### Description

The function `dh_get_p_len` returns the length of the binary representation of prime (**p**) data in Modular Exponential (MODP) algorithm DH key pair `DH_KEY`.

#### Parameter

- `dh`: The pointer to the DH key pair `DH_KEY`.

#### Return

- 0 or a negative value if the generation fails for any reason.
- The length of the binary representation of prime (**p**) data if the function succeeds.

## 6.2.4 Get MODP DH Generator Data Length

```
int dh_get_g_len(  
    DH_KEY* dh);
```

### Description

The function `dh_get_g_len` returns the length of the binary representation of generator (*g*) data in Modular Exponential (MODP) algorithm DH key pair `DH_KEY`.

### Parameter

- `dh`: The pointer to the DH key pair `DH_KEY`.

### Return

- 0 or a negative value if the generation fails for any reason.
- The length of the binary representation of generator (*g*) data if the function succeeds.

## 6.2.5 Get DH Public Key Data String

```
int dh_get_pub_key_data(
    DH_KEY*           dh,
    unsigned char*     buf,
    size_t             len,
    size_t*            key_len);
```

### Description

The function `dh_get_pub_key_data` returns the octet string of DH public key data in most significant byte first (MSB) format.

### Parameter

- `dh`: The pointer to the DH key pair `DH_KEY` which contains the public key.
- `buf`: The data buffer written with the DH public key data string.
- `len`: The length of the data buffer.
- `key_len`: The buffer written with the actual length of DH public key data string.

### Return

- 0 or error code if the generation fails for any reason.
- 1 if the function succeeds.

## 6.2.6 Get MODP DH Prime Data String

```
int dh_get_p_data(
    DH_KEY*           dh,
    unsigned char*     buf,
    size_t             len,
    size_t*            p_len);
```

### Description

The function `dh_get_pub_key_data` returns the octet string of DH prime (**p**) data in MSB format.

### Parameter

- `dh`: The pointer to the DH key pair `DH_KEY`.
- `buf`: The data buffer written with the DH prime (**p**) data string.
- `len`: The length of the data buffer.
- `key_len`: The buffer written with the actual length of DH prime (**p**) data string.

### Return

- 0 or error code if the generation fails for any reason.
- 1 if the function succeeds.

## 6.2.7 Get MODP DH Generator Data String

```
int dh_get_g_data(
    DH_KEY*           dh,
    unsigned char*     buf,
    size_t             len,
    size_t*            g_len);
```

### Description

The function `dh_get_pub_key_data` returns the octet string of DH generator (*g*) data in MSB format.

### Parameter

- `dh`: The pointer to the DH key pair `DH_KEY`.
- `buf`: The data buffer written with the DH generator (*g*) data string.
- `len`: The length of the data buffer.
- `key_len`: The buffer written with the actual length of DH generator (*g*) data string.

### Return

- 0 or error code if the generation fails for any reason.
- 1 if the function succeeds.

## 6.2.8 Helper Functions

### 6.2.8.1 Create DH Key Pair

```
DH_KEY* dh_key_new(void);
```

#### Description

The function `dh_key_new` allocates, initializes and returns an empty DH key pair structure.

#### Parameter

- No input parameter.

#### Return

- `NULL` if an error occurs.
- The address of the new key pair if the function succeeds.

### 6.2.8.2 Free DH Key Pair

```
void dh_key_free(  
    DH_KEY* dhkey);
```

#### Description

The function `dh_key_free` cleans up a key pair passed in `dhkey` and frees up the spaces allocated to it.

#### Parameter

- `dhkey`: A pointer to the key pair. If it is `NULL`, the function will do nothing.

#### Return

- No return

### 6.2.8.3 Create DH Public Key

```
DH_PUB_KEY* dh_pub_key_new(void);
```

#### Description

The function `dh_pub_key_new` allocates, initializes and returns an empty DH public key.

#### Parameter

- No input parameter.

#### Return

- `NULL` if an error occurs.
- The address of the new DH public key if the function succeeds.

### 6.2.8.4 Free DH Public Key

```
void dh_pub_key_free(  
    DH_PUB_KEY*           pubkey);
```

#### Description

The function `dh_pub_key_free` cleans up a DH public key passed in `pubkey` and frees up the spaces allocated to it.

#### Parameter

- `pubkey`: A pointer to the public key. If it is `NULL`, the function will do nothing.

#### Return

- No return

# 7 Asymmetric Digest Signature and Digest Verification

The following sections specify the data types and functions for asymmetric digest signature and digest verification. The asymmetric digest signature signs the digital signatures in Diffie-Hellman handshake messages to be sent. The asymmetric digest verification verifies the digital signatures in received DH handshake messages. Both of the two operations are invoked in DDS Security Authentication Plugin *DDS:Auth:PKI-DH*<sup>[5]</sup>.

## 7.1 Data Types

### 7.1.1 Digest Signature Context Handle

An ASYM\_DIGEST\_SIGN\_CTX is a handle on the context of a digest signature or digest verification operation. It contains the necessary information and objects for signature or verification.

The ASYM\_DIGEST\_SIGN\_CTX context handle can contain a reference to the buffer containing temporary hashed data generated in `asym_digest_sign_update`.

The content of this handle is *implementation-defined*.

## 7.2 Generic Functions

Operations required in asymmetric digest signature and verification are shown in Table 7-1.

TABLE 7-1 ASYMMETRIC DIGEST SIGNATURE AND VERIFICATION OPERATIONS

Operations	Descriptions
<code>asym_digest_sign_ctx_init</code>	Initialize a context handle ASYM_DIGEST_SIGN_CTX
<code>asym_digest_sign_ctx_cleanup</code>	Clean up a context handle ASYM_DIGEST_SIGN_CTX
<code>asym_digest_sign_ctx_new</code>	Create a context handle ASYM_DIGEST_SIGN_CTX
<code>asym_digest_sign_ctx_free</code>	Free a context handle ASYM_DIGEST_SIGN_CTX
<code>asym_digest_sign_sha256_init</code>	Initialize the digest signature with SHA-256
<code>asym_digest_sign_update</code>	Hash the data for later digest signature.
<code>asym_digest_sign_get_size</code>	Get the length of the signature to be generated.
<code>asym_digest_sign_final</code>	Execute digest signature
<code>asym_digest_verf_sha256_init</code>	Initialize the digest verification with SHA-256
<code>asym_digest_verf_update</code>	Hash the data for later digest verification.
<code>asym_digest_verf_final</code>	Execute digest verification

Functions `asym_digest_sign_ctx_init` or `asym_digest_sign_ctx_new` should be called to get ASYM\_DIGEST\_SIGN\_CTX ready before digest signature or verification starts. After digest signature or verification completes, the context handle ASYM\_DIGEST\_SIGN\_CTX should be cleanup by `asym_digest_sign_ctx_cleanup` or be freed by `asym_digest_sign_ctx_free`.

The function `asym_digest_sign_SHA_init` should associate the local private key with current digest signature operation in corresponding hash algorithm **SHA**. Then the data should be hashed with `asym_digest_sign_update`. The length of the buffer for the signature should be determined by calling `asym_digest_sign_get_size`, before signing the hashed data in `asym_digest_sign_final`.

To verify a digest message, the function `asym_digest_verf_SHA_init` should associate the remote public key with current digest verification in corresponding hash algorithm **SHA**. Then the data should be hashed with `asym_digest_verf_update`, before calling `asym_digest_verf_final` to verify the received hashed signature.

## 7.2.1 Operations of Digest Signature Context Handle

### 7.2.1.1 Digest Signature Context Handle Initialization

```
void asym_digest_sign_ctx_init(  
                                ASYM_DIGEST_SIGN_CTX*      ctx);
```

#### Description

The function `asym_digest_sign_ctx_init` initializes a `ASYM_DIGEST_SIGN_CTX` passed in `ctx`.

#### Parameter

- `ctx`: A pointer to the digest signature context handle. It must not be `NULL`.

#### Return

- No return

### 7.2.1.2 Digest Signature Context Handle Cleanup

```
void asym_digest_sign_ctx_cleanup(  
                                 ASYM_DIGEST_SIGN_CTX*      ctx);
```

#### Description

The function `asym_digest_sign_ctx_cleanup` cleans up a `ASYM_DIGEST_SIGN_CTX` passed in `ctx`.

#### Parameter

- `ctx`: A pointer to the digest signature context handle. If it is `NULL`, the function will do nothing.

#### Return

- No return

### 7.2.1.3 Create Digest Signature Context Handle

```
ASYM_DIGEST_SIGN_CTX* asym_digest_sign_ctx_new(void);
```

#### Description

The function `asym_digest_sign_ctx_new` allocates, initializes and returns a `ASYM_DIGEST_SIGN_CTX`.

#### Parameter

- No input parameter.

#### Return

- `NULL` if an error occurs.
- The address of `ASYM_DIGEST_SIGN_CTX` if the function succeeds.

#### 7.2.1.4 Free Digest Signature Context Handle

```
void asym_digest_sign_ctx_free(  
    ASYM_DIGEST_SIGN_CTX*           ctx);
```

##### Description

The function `asym_digest_sign_ctx_free` cleans up a `ASYM_DIGEST_SIGN_CTX` passed in `ctx` and frees up the spaces allocated to it.

##### Parameter

- `ctx`: A pointer to the digest signature context handle. If it is `NULL`, the function will do nothing.

##### Return

- No return

## 7.2.2 Digest Signature Operation Initialization

### 7.2.2.1 Initialization of Digest Signature Operation with SHA-256

```
int asym_digest_sign_sha256_init(  
    ASYM_DIGEST_SIGN_CTX*           ctx,  
    PRIV_KEY*                      pkey);
```

#### Description

The function `asym_digest_sign_sha256_init` prepares for a following digest signature operation. It should setup the configurations in digest signature context handle `ASYM_DIGEST_SIGN_CTX` for a following digest signature operation, such as digest algorithm according to `PRIV_KEY` passed in `pkey`.

SHA-256 should be used as hashed function in following digest operation. This property can be set in function `asym_digest_sign_sha256_init`.

#### Parameter

- `ctx`: A pointer to a digest signature context handle. It must not be `NULL`.
- `pkey`: A pointer to a private key.

#### Return

- Error code if the initialization fails for any reason.
- 1 if the initialization succeeds.

### 7.2.3 Digest Signature Update

```
int asym_digest_sign_update(
    ASYM_DIGEST_SIGN_CTX*           ctx,
    const void*                     data,
    const size_t                     datalen);
```

#### Description

The function `asym_digest_sign_update` hashes `datalen` bytes of data at `data` into the context handle `ctx` for later digest signature. The reference to the hashed data should be set in `ctx` context handle.

The hash function should provide the correct hash algorithm specified in the previously called initialization function listed in section 7.2.2.

#### Parameter

- `ctx`: A pointer to a digest signature context handle. It must not be `NULL`.
- `data`: Reference to an input buffer containing input message.
- `datalen`: The size of message.

#### Return

- `0` if the signature fails for any reason.
- `1` if the signature succeeds.

## 7.2.4 Get Size of Buffer for Digest Signature

```
int asym_digest_sign_get_size(
    ASYM_DIGEST_SIGN_CTX*           ctx,
    size_t*                         signlen);
```

### Description

The function `asym_digest_sign_get_size` determines the length `signlen` of the buffer written with the signature generated in `asym_digest_sign_final`.

The buffer used in `asym_digest_sign_final` to contain the signature should be allocated according to the length `signlen`.

### Parameter

- `ctx`: A pointer to a digest signature context handle. It must not be `NULL`.
- `signlen`: The size of the signature.

### Return

- `0` if the operation fails for any reason.
- `1` if the operation succeeds.

## 7.2.5 Digest Signature Final

```
int asym_digest_sign_final(
    ASYM_DIGEST_SIGN_CTX*           ctx,
    void*                           sign,
    size_t*                         sginlen);
```

### Description

The function `asym_digest_sign_final` signs a message digest contained in the context handle at `ctx` within an asymmetric operation.

Note that only an already hashed message can be signed. Hash function `asym_digest_sign_update` should be invoked before the function starts.

The `sign` buffer length should be previously determined in `asym_digest_sign_get_size`.

### Parameter

- `ctx`: A pointer to a digest signature context handle. It must not be `NULL`.
- `sign`: Reference to an output buffer written with the signature of the digest.
- `sginlen`: The size of the signature.

### Return

- 0 if the signature fails for any reason.
- 1 if the signature succeeds.

## 7.2.6 Digest Verification Operation Initialization

### 7.2.6.1 Initialization of Digest Verification Operation with SHA-256

```
int asym_digest_verf_sha256_init(  
    ASYM_DIGEST_SIGN_CTX*      ctx,  
    PUB_KEY*                  pubkey);
```

#### Description

The function `asym_digest_verf_sha256_init` prepares for a following digest verification operation. It should setup the configurations in digest signature context handle `ASYM_DIGEST_SIGN_CTX` for a following digest verification operation, such as digest algorithm according to `PUB_KEY` passed in `pubkey`.

SHA-256 will be used as hashed function in following digest operation. This property can be set in function `asym_digest_verf_sha256_init`.

#### Parameter

- `ctx`: A pointer to a digest verification context handle. It must not be `NULL`.
- `pub_key`: A pointer to a public key.

#### Return

- 0 if the initialization fails for any reason.
- 1 if the initialization succeeds.

## 7.2.7 Digest Verification Update

```
int asym_digest_verf_update(
    ASYM_DIGEST_SIGN_CTX*           ctx,
    const void*                     data,
    const size_t                     datalen);
```

### Description

The function `asym_digest_verf_update` hashes `datalen` bytes of data at `data` into the context handle `ctx` for later digest verification.

The hash function should provide the correct hash algorithm specified in the previously called initialization function listed in section 7.2.2.

### Parameter

- `ctx`: A pointer to a digest signature context handle. It must not be `NULL`.
- `data`: Reference to an input buffer containing input message.
- `datalen`: The size of message.

### Return

- 0 if the signature fails for any reason.
- 1 if the signature succeeds.

## 7.2.8 Digest Verification Final

```
int asym_digest_verf_final(
    ASYM_DIGEST_SIGN_CTX*           ctx,
    const void*                     sign,
    const size_t                     signlen);
```

### Description

The function `asym_digest_verf_final` verifies a message digest signature within an asymmetric operation.

Note that only an already hashed message can be passed in for verification. Hash function `asym_digest_verf_update` should be invoked before the function starts.

### Parameter

- `ctx`: A pointer to a digest verification context handle. It must not be `NULL`.
- `sign`: Reference to an input buffer containing the signature to be verified.
- `signlen`: The size of the signature.

### Return

- 0 if the verification fails for any reason.
- 1 if the verification succeeds.

## **8 SharedSecret Derivation**

The following sections specify the data types and functions for *SharedSecret* derivation. The following functions derive the *SharedSecret* from local DH private key and remote DH public key in DDS Security Authentication Plugin *DDS:Auth:PKI-DH*<sup>[5]</sup>.

### **8.1 Data Types**

#### **8.1.1 Derivation Context Handle**

A SS\_DERIVE\_CTX is a handle on the context of a *SharedSecret* derivation operation.

The content of this handle is *implementation-defined*.

## 8.2 Generic Functions

*SharedSecret* derivation requires the operation shown in Table 8-1.

TABLE 8-1 *SHAREDSECRET* DERIVATION OPERATIONS

Operations	Descriptions
<code>ss_derive_ctx_new</code>	Create a context handle <code>SS_DERIVE_CTX</code>
<code>ss_derive_ctx_free</code>	Free a context handle <code>SS_DERIVE_CTX</code>
<code>ss_derive_init</code>	Initialize the <i>SharedSecret</i> derivation operation with local DH key.
<code>ss_derive_set_peer</code>	Set the remote public key for derivation operation.
<code>ss_derive_get_size</code>	Get the length of the buffer to contain <i>SharedSecret</i> .
<code>ss_derive_final</code>	Derive the <i>SharedSecret</i> .

Functions `ss_derive_ctx_new` should be called to get `SS_DERIVE_CTX` ready before derivation. After derivation completes, the context handle `SS_DERIVE_CTX` should be freed by `ss_derive_ctx_free`.

The function `ss_derive_set_peer` should setup remote DH public key respectively, before derivation operation.

The length of the buffer to contain should be determined in `ss_derive_get_size`, before `ss_derive_final` derives the *SharedSecret* and put it into that buffer.

## 8.2.1 Operations of Derivation Context Handle

### 8.2.1.1 Create Derivation Context Handle

```
SS_DERIVE_CTX* ss_derive_ctx_new(  
    DH_KEY* key);
```

#### Description

The function `ss_derive_ctx_new` allocates, initializes and returns a `SS_DERIVE_CTX`.

The function should associate with the DH private key of local Participant for derivation contained in DH key pair passed in `key`.

#### Parameter

- `key`: Reference to the DH key pair whose private key is required for derivation.

#### Return

- `NULL` if an error occurs.
- The address of `SS_DERIVE_CTX` if the function succeeds.

### 8.2.1.2 Free Derivation Context Handle

```
void ss_derive_ctx_free(  
    SS_DERIVE_CTX* ctx);
```

#### Description

The function `ss_derive_ctx_free` cleans up a `SS_DERIVE_CTX` passed in `ctx` and frees up the spaces allocated to it.

#### Parameter

- `ctx`: A pointer to the HMAC context handle. If it is `NULL`, the function will do nothing.

#### Return

- No return

## 8.2.2 Derivation Initialization

```
int ss_derive_init(  
    SS_DERIVE_CTX*           ctx);
```

### Description

The function `ss_derive_init` initializes a derivation represented by a context handle passed in `ctx`.

### Parameter

- `ctx`: A pointer to the context handle. It must not be `NULL`.

### Return

- 0 if the initialization fails for any reason.
- 1 if the initialization succeeds.

### 8.2.3 Set Peer DH Public Key

```
int ss_derive_set_peer_key(
    SS_DERIVE_CTX*           ctx,
    DH_PUB_KEY*               pub_key);
```

#### Description

The function `ss_derive_set_peer` setup the DH public key from remote matched Participant for the derivation. The DH public key is received from DH handshake message exchange.

#### Parameter

- `ctx`: A pointer to the context handle. It must not be `NULL`.
- `pub_key`: Reference to the DH public key required for derivation.

#### Return

- 0 if the setup fails for any reason.
- 1 if the setup succeeds.

## 8.2.4 Get the Size of *SharedSecret*

```
int ss_derive_get_size(
    SS_DERIVE_CTX*           ctx,
    size_t*                   sslen);
```

### Description

The function `ss_derive_get_size` determines the length `sslen` of the buffer written with the *SharedSecret* generated in `ss_derive_final`.

The buffer used in `ss_derive_final` to contain the *SharedSecret* should be allocated according to the length `sslen`.

### Parameter

- `ctx`: A pointer to a digest verification context handle. It must not be `NULL`.
- `sslen`: The size of the *SharedSecret*.

### Return

- `0` if the derivation fails for any reason.
- `1` if the derivation succeeds.

## 8.2.5 Derivation Execution

```
int ss_derive_final(
    SS_DERIVE_CTX*           ctx,
    void*                     ss,
    size_t*                   sslen);
```

### Description

The function `ss_derive_final` takes asymmetric derivation operation parameters set in above functions as input, and outputs a key object.

### Parameter

- `ctx`: A pointer to a digest verification context handle. It must not be `NULL`.
- `ss`: Reference to an output buffer written with generated *SharedSecret*.
- `sslen`: The size of the *SharedSecret*.

### Return

- 0 if the derivation fails for any reason.
- 1 if the derivation succeeds.

# 9 Hash-Based Message Authentication Code

The following sections describes the data types and functions for Hash-Based Message Authentication Code (HMAC). HMAC is used to create `master_salt`, `master_sender_key` and `SessionKey` in DDS Security built-in Cryptographic plugin `DDS: Crypto: AES-GCM-GMAC[5]`.

## 9.1 Data Types

### 9.1.1 HMAC Key

A `HMAC_KEY` represent a secret key used to generate MAC in HMAC.

The content of this handle is *implementation-defined*.

### 9.1.2 HMAC Context Handle

A `HAMC_CTX` is a hander to the context of HAMC operation. It should contain necessary information for HMAC operation.

The content of this handle is *implementation-defined*.

## 9.2 Generic Functions

HMAC operations are listed in Table 9-1.

TABLE 9-1 HMAC OPERATIONS

Operations	Descriptions
<code>hmac_key_new</code>	Create and initialize a HMAC key.
<code>hmac_key_free</code>	Free a HMAC key.
<code>hmac_ctx_init</code>	Initialize a context handle <code>HMAC_CTX</code>
<code>hmac_ctx_cleanup</code>	Clean up a context handle <code>HMAC_CTX</code>
<code>hmac_ctx_new</code>	Create a context handle <code>HMAC_CTX</code>
<code>hmac_ctx_free</code>	Free a context handle <code>HMAC_CTX</code>
<code>hmac_sha256_init</code>	Initialize HMAC SHA-256 operation.
<code>hmac_update</code>	Feed the data to HMAC operation
<code>hmac_compute_final</code>	Execute the HMAC operation and compute the HMAC

Functions `hmac_ctx_init` or `hmac_ctx_new` should be called to get `HMAC_CTX` ready before HMAC operation starts. After HMAC completes, the context handle `HMAC_CTX` should be cleanup by `hmac_ctx_cleanup` or be freed by `hmac_ctx_free`.

The function `hmac_SHA_init` should setup hash operation with corresponding hash algorithm `SHA`. After the HMAC initialization completes, `hmac_update` accumulates the data which is used by the function `hmac_compute_final` to finally compute the MAC.

## 9.2.1 Operations of HMAC Key

### 9.2.1.1 Create a HMAC Key

```
HMAC_KEY* hmac_key_new(  
    const unsigned char*     key,  
    size_t                   len);
```

#### Description

The function `hmac_key_new` allocates, initializes and returns a HMAC key `HMAC_KEY`. The key data passed in `key` in length `len` is set into the HMAC key object.

#### Parameter

- `key`: A reference to the secret key data used in MAC.
- `len`: length of the key data.

#### Return

- `NULL` if an error occurs.
- The address of `HMAC_KEY` if the function succeeds.

### 9.2.1.2 Free a HMAC Key

```
void hmac_key_free(  
    HMAC_KEY*               key);
```

#### Description

The function `hmac_ctx_free` cleans up a HMAC key passed in `key` and frees up the spaces allocated to it.

#### Parameter

- `key`: A pointer to the HMAC key. If it is `NULL`, the function will do nothing.

#### Return

- No return

## 9.2.2 Operations of HMAC Context Handle

### 9.2.2.1 HMAC Context Handle Initialization

```
void hmac_ctx_init(  
                    HMAC_CTX*          ctx);
```

#### Description

The function `hmac_ctx_init` initializes a `HMAC_CTX` passed in `ctx`.

The specific initialization is *implementation-defined*.

#### Parameter

- `ctx`: A pointer to the HMAC context handle. It must not be `NULL`.

#### Return

- No return

### 9.2.2.2 HMAC Context Handle Cleanup

```
void hmac_ctx_cleanup(  
                      HMAC_CTX*          ctx);
```

#### Description

The function `hmac_ctx_cleanup` cleans up a `HMAC_CTX` passed in `ctx`.

The specific initialization is *implementation-defined*.

#### Parameter

- `ctx`: A pointer to the HMAC context handle. If it is `NULL`, the function will do nothing.

#### Return

- No return

### 9.2.2.3 Create HMAC Context Handle

```
HMAC_CTX* hmac_ctx_new(void);
```

#### Description

The function `hmac_ctx_new` allocates, initializes and returns a context handle `HMAC_CTX`.

#### Parameter

- No input parameter.

#### Return

- `NULL` if an error occurs.
- The address of `HMAC_CTX` if the function succeeds.

#### 9.2.2.4 Free HMAC Context Handle

```
void hmac_ctx_free(  
    HMAC_CTX* ctx);
```

##### Description

The function `hmac_ctx_free` cleans up a `HMAC_CTX` passed in `ctx` and frees up the spaces allocated to it.

##### Parameter

- `ctx`: A pointer to the HMAC context handle. If it is `NULL`, the function will do nothing.

##### Return

- No return

## 9.2.3 HMAC Operation Initialization

### 9.2.3.1 HMAC with SHA-256 Initialization

```
int hmac_sha256_init(  
    HMAC_CTX*           ctx,  
    HMAC_KEY*            key);
```

#### Description

The function `hmac_sha256_init` initializes a HMAC operation represent by a HAMC context handle passed in `ctx`. A HMAC key to be associated with HMAC operation is passed in `key`.

SHA-256 is used in function `hmac_sha256_init`.

The function `hmac_update` should be called after the function `hmac_sha256_init` completes the initialization.

#### Parameter

- `ctx`: A pointer to a HMAC context handle. It must not be `NULL`.
- `key`: A pointer to the input HMAC key to be associated with HMAC operation. It must not be `NULL`.

#### Return

- 0 if the initialization fails for any reason.
- 1 if the initialization succeeds.

## 9.2.4 HMAC Update

```
int hmac_update(  
    HMAC_CTX*           ctx,  
    const void*          chunk,  
    size_t               len);
```

### Description

The function `hmac_update` accumulates data passed in `chunk` for a HMAC calculation.

The function `hmac_update` should be called after the function `hmac_sha256_init` completes the initialization.

### Parameter

- `ctx`: A pointer to a HMAC context handle. It must not be `NULL`.
- `chunk`: A buffer pointer to the data to be hashed. It must not be `NULL`.
- `len`: The size of `chunk`.

### Return

- 0 if the update fails for any reason.
- 1 if the update succeeds.

## 9.2.5 HMAC Update

```
int hmac_get_size(  
    HMAC_CTX* ctx,  
    size_t* len);
```

### Description

The function `hmac_get_size` returns the length of HMAC result in `len`.

The function `hmac_get_size` should be called before the function `hmac_compute_final` to determine the HMAC output buffer size.

### Parameter

- `ctx`: A pointer to a HMAC context handle. It must not be `NULL`.
- `len`: The length of HMAC output.

### Return

- 0 if the operation fails for any reason.
- 1 if the operation succeeds.

## 9.2.6 HMAC Compute Final

```
int hmac_compute_final(
    HMAC_CTX* ctx,
    void* mac,
    size_t* len);
```

### Description

The function `hmac_compute_final` finalizes the HMAC operation and computes the HMAC.

### Parameter

- `ctx`: A pointer to a HMAC context handle. It must not be `NULL`.
- `mac`: A pointer to the output buffer to be filled with the computed HMAC. It must not be `NULL`. The buffer size of `mac` should be enough to hold HMAC result.
- `len`: the buffer contains the actual length of computed HMAC result.

### Return

- 0 if the HMAC fails for any reason.
- 1 if the HMAC succeeds.

# 10 Authenticated Encryption and Decryption

The following sections describes the data types and functions for Authenticated Encryption and Decryption (AE). Authentication Encryption encrypts a message and create its MAC. Authentication Decryption verifies a MAC and decrypts the corresponding message if the MAC verification passes. AE is used in symmetric keys exchange and RTPS message communication in DDS Security built-in Cryptographic plugin *DDS: Crypto: AES-GCM-GMAC*<sup>[5]</sup>.

## 10.1 Data Types

### 10.1.1 AE Context Handle

An `AE_CTX` is a handle to the context of AE operation. It should contain necessary information for AE operation.

The content of this handle is *implementation-defined*.

### 10.1.2 AES Block Size

`AES_BLOCK_SIZE` specifies the Advanced Encryption Standard (AES) block size in bytes.

## 10.2 Generic Functions

AE operations are listed in Table 10-1.

TABLE 10-1 AUTHENTICATION ENCRYPTION AND DECRYPTION OPERATIONS

Operations	Descriptions
<code>ae_ctx_init</code>	Initialize a context handle <code>AE_CTX</code>
<code>ae_ctx_cleanup</code>	Clean up a context handle <code>AE_CTX</code>
<code>ae_ctx_new</code>	Create a context handle <code>AE_CTX</code>
<code>ae_ctx_free</code>	Free a context handle <code>AE_CTX</code>
<code>ae_encrypt_aesgcm_128_init</code>	Initialize AES-128-GCM authentication encryption.
<code>ae_decrypt_aesgcm_128_init</code>	Initialize AES-128-GCM authentication decryption.
<code>ae_encrypt_aesgcm_256_init</code>	Initialize AES-256-GCM authentication encryption.
<code>ae_decrypt_aesgcm_256_init</code>	Initialize AES-256-GCM authentication decryption.
<code>ae_encrypt_update_aad</code>	Update Additional Authenticated Data (AAD) for GMAC authentication encryption.
<code>ae_decrypt_update_aad</code>	Update AAD for GMAC authentication decryption.
<code>ae_encrypt_update</code>	Update data for GCM authentication encryption.
<code>ae_decrypt_update</code>	Update data for GCM authentication decryption.
<code>ae_encrypt_final</code>	Finalize authentication encryption.
<code>ae_decrypt_final</code>	Finalize authentication decryption.

Functions `ae_ctx_init` or `ae_ctx_new` should be called to get `AE_CTX` ready before AE operation starts. After AE completes, the context handle `HMAC_CTX` should be cleanup by `ae_ctx_cleanup` or be freed by `ae_ctx_free`.

The function `ae_encrypt_aesgcm_KEY_init` should setup authentication encryption operation with corresponding AES-GCM/AES-GMAC keys in the specific length `KEY`. Correspondingly, The function `ae_decrypt_aesgcm_KEY_init` should setup authentication decryption operation with corresponding AES-GCM/AES-GMAC keys in the specific length `KEY`.

The function `ae_encrypt_update` accumulates the data for AE operation. If AAD is required, `ae_encrypt_update_aad` should also be invoked to append the AAD. The function `ae_encrypt_final` finally encrypts the data and calculates the authentication tag.

Similarly, the function `ae_decrypt_update` accumulates the data for decryption operation. If AAD is required, `ae_decrypt_update_aad` should also be invoked to append the AAD. The function `ae_decrypt_final` finally decrypts the received data and authenticate the tag.

## 10.2.1 Operations of AE Context Handle

### 10.2.1.1 AE Context Handle Initialization

```
void ae_ctx_init(  
                  AE_CTX*           ctx);
```

#### Description

The function `ae_ctx_init` initializes a `AE_CTX` passed in `ctx`.

The specific initialization is *implementation-defined*.

#### Parameter

- `ctx`: A pointer to the AE context handle. It must not be `NULL`.

#### Return

- No return

### 10.2.1.2 AE Context Handle Cleanup

```
void ae_ctx_cleanup(  
                     AE_CTX*           ctx);
```

#### Description

The function `ae_ctx_cleanup` cleans up a `AE_CTX` passed in `ctx`.

The specific initialization is *implementation-defined*.

#### Parameter

- `ctx`: A pointer to the AE context handle. If it is `NULL`, the function will do nothing.

#### Return

- No return

### 10.2.1.3 Create AE Context Handle

```
AE_CTX* ae_ctx_new(void);
```

#### Description

The function `ae_ctx_new` allocates, initializes and returns a context handle `AE_CTX`.

#### Parameter

- No input parameter.

#### Return

- `NULL` if an error occurs.
- The address of `AE_CTX` if the function succeeds.

#### 10.2.1.4 Free AE Context Handle

```
void ae_ctx_free(  
    AE_CTX* ctx);
```

##### Description

The function `ae_ctx_free` cleans up a `AE_CTX` passed in `ctx` and frees up the spaces allocated to it.

##### Parameter

- `ctx`: A pointer to the AE context handle. If it is `NULL`, the function will do nothing.

##### Return

- No return

## 10.2.2 AE Operation Initialization

### 10.2.2.1 Authentication Encryption Initialization with AES-128-GCM

```
int ae_encrypt_aesgcm_128_init(
    AE_CTX*                 ctx,
    const void*              key,
    const void*              iv);
```

#### Description

The function `ae_encrypt_aesgcm_128_init` initializes the Authentication Encryption operation using AES-128-GCM. It setups the key passed in `key` and the Initialization Vectors specified by `iv`. The encoding of `iv` should follow DDS security specification<sup>[5]</sup>.

#### Parameter

- `ctx`: A pointer to a AE context handle.
- `key`: A pointer to the input key data for AE operation.
- `iv`: A pointer to the Initialization Vectors.

#### Return

- 0 if the initialization fails for any reason.
- 1 if the initialization succeeds.

### 10.2.2.2 Authentication Decryption Initialization with AES-128-GCM

```
int ae_decrypt_aesgcm_128_init(
    AE_CTX*                 ctx,
    const void*              key,
    const void*              iv);
```

#### Description

The function `ae_decrypt_aesgcm_128_init` initializes the Authentication Decryption operation using AES-128-GCM. It setups the key passed in `key` and the Initialization Vectors specified by `iv`. The encoding of `iv` should follow DDS security specification<sup>[5]</sup>.

#### Parameter

- `ctx`: A pointer to a AE context handle.
- `key`: A pointer to the input key data.
- `iv`: A pointer to the Initialization Vectors.

#### Return

- 0 if the initialization fails for any reason.
- 1 if the initialization succeeds.

### 10.2.2.3 Authentication Encryption Initialization with AES-256-GCM

```
int ae_encrypt_aesgcm_256_init(
```

```
AE_CTX* ctx,  
const void* key,  
const void* iv);
```

## Description

The function `ae_encrypt_aesgcm_256_init` initializes the Authentication Encryption operation using AES-128-GCM. It setups the key passed in `key` and the Initialization Vectors specified by `iv`. The encoding of `iv` should follow DDS security specification<sup>[5]</sup>.

## Parameter

- `ctx`: A pointer to a AE context handle.
- `key`: A pointer to the input key data for AE operation.
- `iv`: A pointer to the Initialization Vectors.

## Return

- 0 if the initialization fails for any reason.
- 1 if the initialization succeeds.

### 10.2.2.4 Authentication Decryption Initialization with AES-256-GCM

```
int ae_decrypt_aesgcm_256_init(  
    AE_CTX* ctx,  
    const void* key,  
    const void* iv);
```

## Description

The function `ae_decrypt_aesgcm_256_init` initializes the Authentication Decryption operation using AES-256-GCM. It setups the key passed in `key` and the Initialization Vectors specified by `iv`.

The encoding of `iv` should follow DDS security specification<sup>[5]</sup>.

## Parameter

- `ctx`: A pointer to a AE context handle.
- `key`: A pointer to the input key data.
- `iv`: A pointer to the Initialization Vectors.

## Return

- 0 if the initialization fails for any reason.
- 1 if the initialization succeeds.

## 10.2.3 AE Additional Authenticated Data Update

The following functions add Additional Authenticated Data (AAD) to AE operations.

### 10.2.3.1 Authenticated Encryption AAD Update

```
int ae_encrypt_update_aad(
    AE_CTX*                  ctx,
    const void*                aad,
    size_t                     len);
```

#### Description

The function `ae_encrypt_update_aad` feeds a new chunk of Additional Authenticated Data (AAD) passed in `aad` to the Authentication Encryption operation.

#### Parameter

- `ctx`: A pointer to a AE context handle.
- `aad`: A pointer to the input AAD for AE operation.
- `len`: The size of `aad` in bytes.

#### Return

- 0 if the AAD update fails for any reason.
- 1 if the AAD update succeeds.

### 10.2.3.2 Authenticated Decryption AAD Update

```
int ae_decrypt_update_aad(
    AE_CTX*                  ctx,
    const void*                aad,
    size_t                     len);
```

#### Description

The function `ae_decrypt_update_aad` feeds a new chunk of AAD passed in `aad` to the Authentication Decryption operation.

#### Parameter

- `ctx`: A pointer to a AE context handle.
- `aad`: A pointer to the input AAD.
- `len`: The size of `aad` in bytes.

#### Return

- 0 if the AAD update fails for any reason.
- 1 if the AAD update succeeds.

## 10.2.4 AE Data Update

### 10.2.4.1 Authenticated Encryption Data Update

```
int ae_encrypt_update(
    AE_CTX*           ctx,
    void*             out,
    int*              outlen,
    const void*       in,
    size_t            inlen);
```

#### Description

The function `ae_encrypt_update` encrypts `inlen` bytes from the buffer `in` and writes the encrypted version to `out`.

#### Parameter

- `ctx`: A pointer to a AE context handle.
- `out`: the buffer written with cipher text.
- `outlen`: the length of the cipher text in bytes.
- `in`: A pointer to the input data for encryption.
- `inlen`: The size of the data for encryption in bytes.

#### Return

- 0 if the update fails for any reason.
- 1 if the update succeeds.

### 10.2.4.2 Authenticated Decryption Data Update

```
int ae_decrypt_update(
    AE_CTX*           ctx,
    void*             out,
    int*              outlen,
    const void*       in,
    size_t            inlen);
```

#### Description

The function `ae_decrypt_update` decrypts `inlen` bytes from the buffer `in` and writes the plaintext version to `out`.

#### Parameter

- `ctx`: A pointer to a AE context handle.
- `out`: buffer written with the output plain text.
- `outlen`: the length of the output plain text in bytes.
- `in`: A pointer to the input data for decryption.
- `inlen`: The size of the data for decryption in bytes.

#### Return

- 0 if the update fails for any reason.

- 1 if the update succeeds.

## 10.2.5 AE Encryption Final

```
int ae_encrypt_final(
    AE_CTX*           ctx,
    void*              buf,
    int*               buflen,
    void*              tag,
    size_t              taglen);
```

### Description

The function `ae_encrypt_final` completes Authenticated Encryption operation and computes the tag.

### Parameter

- `ctx`: A pointer to a AE context handle.
- `buf`: A pointer to the output data buffer for cipher text. If only AES-GMAC authentication tag is required, `buf` can be omitted. Otherwise, it must not be `NULL`.
- `buflen`: The size of cipher text in bytes.
- `tag`: A pointer to the output authentication tag buffer. It must not be `NULL`.
- `taglen`: The size of authentication tag in bytes.

### Return

- 0 if the encryption or computing tag fails for any reason.
- 1 if the entire process succeeds.

## 10.2.6 AE Decryption Final

```
int ae_decrypt_final(
    AE_CTX*           ctx,
    void*              buf,
    int*               buflen,
    void*              tag,
    size_t              taglen);
```

### Description

The function `ae_decrypt_final` completes Authenticated Decryption operation and compares the computed tag with the tag supplied in the parameter `tag`.

### Parameter

- `ctx`: A pointer to a AE context handle.
- `buf`: A pointer to the output data buffer for plain text. If only AES-GMAC authentication tag is required, `buf` can be omitted. Otherwise, it must not be `NULL`.
- `buflen`: A reference to the size of plain text in bytes.
- `tag`: A pointer to the input authentication tag to be compared. It must not be `NULL`.
- `taglen`: The size of authentication tag in bytes.

### Return

- 0 if the decryption or the comparison fails for any reason.
- 1 if the decryption and comparison both succeed.

# 11 Random Number Generation

## 11.1 Generic Functions

### 11.1.1 Random Number Generation

```
int generate_rand(  
    void* buf,  
    unsigned int len);
```

#### Description

The function `generate_rand` generates random data in `len` bytes.

#### Parameter

- `buf`: Reference to the generated random data.
- `len`: Byte length of the requested random data.

#### Return

- 0 if the generation fails for any reason.
- 1 if the generation succeeds.

# 12 Large Integers

## 12.1 Data Types

### 12.1.1 BIGNUM

A `BIGNUM` type is a placeholder for the memory structure which represents of a large multi-precision integer.

The specific definition is *implementation-defined*.

## 12.2 Generic Functions

Large integer operations are listed in Table 12-1.

TABLE 12-1 LARGE INTEGER OPERATIONS

Operations	Descriptions
<b>big_num_new</b>	Allocate a new large integer object
<b>big_num_free</b>	Free a large integer object
<b>big_num_bytes</b>	Count the number of bytes in a large integer object.
<b>big_num_octstr2bn</b>	Converts a most significant byte first (MSB) octet string into a large integer object.
<b>big_num_bn2octstr</b>	Converts a positive large integer to an MSB octet string.
<b>big_num_rand</b>	Generates a cryptographically strong pseudo-random number of bits and store them in a BIGNUM.

## 12.2.1 Large Integer Object Allocation

```
BIGNUM* big_num_new(void);
```

### Description

The function `big_num_new` allocate a new `BIGNUM` type object.

### Parameter

- No input parameter.

### Return

- `NULL` if the allocation fails for any reason.
- The pointer to a new `BIGNUM` type object if the function succeeds.

## 12.2.2 Large Integer Object Free

```
void big_num_free(  
    BIGNUM* bn);
```

### Description

The function `big_num_free` clears and frees a `BIGNUM` type object.

### Parameter

- `bn`: Reference to the large integer to be free. If it is `NULL`, the function will do nothing.

### Return

- No return

### 12.2.3 Count Number of Bytes in Large Integer

```
int big_num_bytes(  
    BIGNUM*  
    bn);
```

#### Description

The function `big_num_bytes` returns the number of bytes in the nature binary representation of a `BIGNUM` type object.

#### Parameter

- `bn`: Reference to the large integer.

#### Return

- The number of bytes of the large integer. If the large integer equals to zero, it will return 0.

## 12.2.4 Convert Big Endian String to Large Integer

```
BIGNUM* big_num_octstr2bn(  
    const unsigned char*     str,  
    size_t                  len,  
    BIGNUM*                 bn);
```

### Description

The function `big_num_octstr2bn` converts a `len` byte octet string buffer `str` into a `BIGNUM` format. The octet string is in most significant byte first (MSB) representation. The integer represented in the octet string should be positive.

### Parameter

- `str`: A pointer to the buffer containing the octet string representation of the integer.
- `len`: The length of octet string in bytes.
- `bn`: A pointer to a `BIGNUM` to hold the result. If `bn` equals to `NULL`, `big_num_octstr2bn` will allocate a new `BIGNUM` object.

### Return

- `NULL` if the conversion fails for any reason.
- The address of the `BIGNUM` object if the conversion succeeds.

## 12.2.5 Convert Large Integer to Big Endian String

```
int big_num_bn2octstr(
    const BIGNUM*          bn,
    unsigned char*          str);
```

### Description

The function `big_num_octstr2bn` converts a positive large integer in `BIGNUM` format to an octet string. The octet string is written in most significant byte first (MSB) representation.

### Parameter

- `bn`: A pointer to a `BIGNUM` to be converted to an octet string.
- `str`: Reference to the buffer where the octet string representation of the integer is written.

### Return

- The length of the octet string in bytes.

## 12.2.6 Generate Random Number in Large Integer

```
int big_num_rand(  
    BIGNUM* bn,  
    size_t bits);
```

### Description

Generates a cryptographically strong pseudo-random number of `bits` bits in length and stores it in `bn`.

### Parameter

- `bn`: A pointer to a `BIGNUM` to contain the random number.
- `bits`: length in bits of the random number.

### Return

- 0 if the generation fails for any reason.
- 1 if the generation succeeds.