



MatrixSSL Certificates and Certificate Revocation Lists

Electronic versions are uncontrolled unless directly accessed from the QA Document Control system.

Printed version are uncontrolled except when stamped with 'VALID COPY' in red.

External release of this document may require a NDA.

© INSIDE Secure - 2019 - All rights reserved

TABLE OF CONTENTS

1.1 Overview of tools and APIs	5
1.2 Related documents	6
1.3 Compile-time configuration.....	6
2 MATRIXSSL X.509 API REFERENCE.....	7
2.1 Certificate API	7
2.1.1 psX509Cert_t data type	7
2.1.2 psX509ParseCertFile.....	8
2.1.3 psX509ParseCert	9
2.1.4 psX509FreeCert	10
2.1.5 matrixValidateCertsExt	11
2.1.6 matrixValidateCerts	13
2.1.7 psBase64EncodeAndWrite	13
2.1.8 X.509 Getter API	14
2.1.8.1 psX509GetCertPublicKeyDer	14
2.1.8.2 psX509GetOnlineDN	16
2.1.8.3 psX509GetNumOrganizationalUnits	16
2.1.8.4 psX509GetOrganizationalUnit	16
2.1.8.5 psX509GetNumDomainComponents	16
2.1.8.6 psX509GetDomainComponent	17
2.1.8.7 psX509GetConcatenatedDomainComponent	17
2.1.9 X.509 Generation API.....	17
2.1.9.1 Overview	18
2.1.9.2 Initializing.....	18
2.1.9.3 Memory allocation	18
2.1.9.4 psX509SetDNAttribute.....	18
2.1.9.5 psX509SetValidDays	19
2.1.9.6 psX509SetValidNotBefore	20
2.1.9.7 psX509SetValidNotAfter	20
2.1.9.8 psX509SetSerialNum	21
2.1.9.9 psX509SetCertHashAlg	21
2.1.9.10 psX509AddSubjectAltName.....	22
2.1.9.11 psX509AddIssuerAltName.....	23
2.1.9.12 psX509AddKeyUsageBit.....	23
2.1.9.13 psX509AddExtendedKeyUsage.....	24
2.1.9.14 psX509SetSubjectKeyld	24
2.1.9.15 psX509SetBasicConstraintsCA.....	25
2.1.9.16 psX509SetBasicConstraintsPathLen	25
2.1.9.17 psX509AddAuthorityInfoAccess.....	26
2.1.9.18 psX509AddPolicy	27
2.1.9.19 psX509SetConstraintRequireExplicitPolicy	29
2.1.9.20 psX509SetConstraintInhibitPolicyMappings.....	29
2.1.9.21 psX509AddPolicyMapping	30
2.1.9.22 psX509SetNetscapeComment.....	30
2.1.9.23 psX509SetPublicKey	31
2.1.9.24 psWriteCertReqMem	31
2.1.9.25 psParseCertReqFile	32
2.1.9.26 psParseCertReqBuf	32
2.1.9.27 psParseCertReqBufExt.....	33

2.1.9.28 psCertReqGetSignatureAlgorithm.....	33
2.1.9.29 psCertReqGetPubKeyAlgorithm	33
2.1.9.30 psCertReqGetVersion.....	34
2.1.9.31 psX509SetCAIssuedCertExtensions.....	34
2.1.9.32 psX509WriteCAIssuedCert	35
2.1.9.33 psX509WriteSelfSignedCert	37
2.2 Certificate Revocation List API.....	38
2.2.1 psX509Crl_t data type	38
2.2.2 psX509ParseCRL	38
2.2.3 psCRL_Update	39
2.2.4 psCRL_determineRevokedStatus.....	39
2.2.5 psCRL_Delete	40
2.2.6 psCRL_DeleteAll	40
2.2.7 psCRL_Remove	40
2.2.8 psCRL_RemoveAll	41
2.2.9 psX509FreeCRL	41
2.2.10 psCRL_GetCRLForCert.....	41
2.2.11 psX509GetCRLdistURL.....	41
2.2.12 psX509AuthenticateCRL	42
2.3 Distinguished Name Attributes	42
2.4 Certificate Extensions	43
2.4.1 Supported Extensions.....	43
2.4.2 Accessing information in certificate extensions	44
3 CERTIFICATE REVOCATION LISTS.....	45
3.1 Overview	45
3.2 X.509 Certificates and CRL Distribution Points	45
3.3 Parsing and Authentication of a CRL	45
3.4 MatrixSSL and CRL	45
3.5 Example application CRL support	46
3.6 Example application API	47
3.6.1 fetchCRL	47
3.6.2 fetchParseAndAuthCRLfromCert.....	47
3.6.3 fetchParseAndAuthCRLfromUrl	48
3.7 Supported CRL API.....	49
3.7.1 psX509Crl_t data type	49
3.7.2 psX509ParseCRL	49
3.7.3 psCRL_Update	50
3.7.4 psCRL_determineRevokedStatus.....	50
3.7.5 psCRL_Delete	51
3.7.6 psCRL_DeleteAll	51
3.7.7 psCRL_Remove	51
3.7.8 psCRL_RemoveAll	51
3.7.9 psX509FreeCRL.....	52
3.7.10 psCRL_GetCRLForCert.....	52
3.7.11 psX509GetCRLdistURL.....	52
3.7.12 psX509AuthenticateCRL	52

ABOUT THIS DOCUMENT

MatrixSSL provides an API and a set of utility programs that support common requirements for X.509 certificate handling. These include certificate creation, parsing and encoding, authentication, certificate chain validation and revocation checking.

The MatrixSSL X.509 API is used internally by the SSL library and is mostly transparent to the SSL user. However, the API can also be exploited outside of the SSL use case. This document is geared towards the latter use case.

1.1 Overview of tools and APIs

MatrixSSL provides a public application API for certificate and CRL parsing, certificate validation and revocation checking. Certificate and certificate signing request generation can be performed by utility programs supplied with MatrixSSL or by using the X.509 Generation API described in this document. The following table presents a summary of how some common certificate-related tasks are supported by MatrixSSL.

Task	Support
Generating certificate signing requests (CSRs)	Application API: X.509 Generation API Utility program: apps/crypto/certrequest.c
Generating certificates from CSRs	Application API: X.509 Generation API Utility program: apps/crypto/certgen.c
Generating self-signed certificates	Application API: X.509 Generation API Utility program: apps/crypto/certgen.c
Loading and parsing certificates	Application API: psX509ParseCertFile and psX509ParseCert
Certificate chain validation	Application API: matrixValidateCerts. Utility program: matrixssl/test/certValidate.c
Loading and parsing private keys and certificates for use in SSL connections	Application API: matrixSslLoadRsaKeys, matrixSslLoadEcKeys
Parsing certificate revocation lists (CRLs)	Application API: psX509ParseCRL
Revocation checking	Application API: psCRL_determineRevokedStatus
Downloading CRLs	Example program: apps/ssl/client.c Example API: fetchCRL, fetchParseAndAuthCRLFromCert

1.2 Related documents

This document works in close collaboration with other MatrixSSL documents. It is highly recommended to additionally consult the following documents in order to gain a good understanding of MatrixSSL's X.509 support:

MatrixSSL API	Describes the <code>psX509Cert_t</code> structure that is used to hold parsed X.509 certificates, as well as related structures. Provides information on how to supply a user-provided certificate validation callback for use in SSL connections and how to integrate it with MatrixSSL-provided certificate validation routines.
MatrixSSL Developer Guide	Contains general information on public-key infrastructure, certificate validation and authentication. Describes in more detail the different checks performed in the certificate chain validation process used by the <code>matrixValidateCerts</code> API function.
Matrix Key and Certificate Generation Utilities	Provides usage instructions for the certificate signing request and certificate creation tools.

1.3 Compile-time configuration

MatrixSSL's X.509 features can be enabled by defining the following macros in `cryptoConfig.h`:

USE_X509	Enables minimal X.509 support. Enabling this macro is a pre-condition for other macros listed in this table.
USE_CERT_PARSE	Enables support for X.509 certificate certificate parsing.
USE_FULL_CERT_PARSE	Enable parsing of the following additional certificate extensions: <code>nameConstraints</code> , <code>certificatePolicies</code> , <code>policyConstraints</code> , <code>policyMappings</code> , <code>authorityInfoAccess</code> .
USE_CERT_GEN	Enables generation of X.509 certificates and certificate signing requests. Required for both the command-line generation tools and the X.509 Generation API.
USE_CRL	Enables support for certificate revocation lists.
USE_EXTRA_DN_ATTRIBUTES_RFC5280_SHOULD	Enables support for those distinguished name (DN) attributes which, according to RFC 5280, SHOULD be supported. Note that the DN attributes that MUST be supported according to RFC 5280 are always enabled and to not need a separate define.
USE_EXTRA_DN_ATTRIBUTES	Enables support for extra distinguished name (DN) attributes not mentioned in RFC 5280.

2 MATRIXSSL X.509 API REFERENCE

2.1 Certificate API

2.1.1 psX509Cert_t data type

In MatrixSSL, the psX509Cert_t structure is used to represent parsed X.509 certificates. This structure is described here for quick reference. For more information on this structure, please consult section 5 of the MatrixSSL API manual.

```
typedef struct psCert {
    psPool_t          *pool;
    int32             version;
    unsigned char      *serialNumber;
    uint32            serialNumberLen;
    x509DNAttributes_t issuer;
    x509DNAttributes_t subject;
    int32             notBeforeTimeType;
    int32             notAfterTimeType;
    char              *notBefore;
    char              *notAfter;
    psPubKey_t         publicKey;
    int32             pubKeyAlgorithm;
    int32             certAlgorithm;
    int32             sigAlgorithm;
    unsigned char      *signature;
    uint32            signatureLen;
    unsigned char      sigHash[MAX_HASH_SIZE];
    unsigned char      *uniqueIssuerId;
    uint32            uniqueIssuerIdLen;
    unsigned char      *uniqueSubjectId;
    uint32            uniqueSubjectIdLen;
    x509v3extensions_t extensions;
    int32             authStatus;
    uint32            authFailFlags;
    unsigned char      *unparsedBin;
    uint32            binLen;
    struct psCert      *next;
} psX509Cert_t;
```

2.1.2 psX509ParseCertFile

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

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the output certificate will be allocated
fileName	input	Filename of the PEM format certificate. Multiple files can be loaded at once by providing a list of filenames separated by semicolons.
outcert	output	The output psX509Cert structure. Must be freed by caller if the function is successful.
flags	input	Additional flags. CERT_STORE_UNPARSED_BUFFER: Keep the raw, ASN.1 DER encoded certificate in the unparsedBin member of the psX509Cert structure. CERT_STORE_DN_BUFFER: Store DN Attributes in the psX509Cert structure.

Return Value	Description
> 0	Success. A valid psX509Cert structure is allocated and populated in "outcert" parameter. The value returned is the number of certificates that were successfully parsed.
PS_MEM_FAIL	Failure. Unable to allocate memory for the structure
PS_PARSE_FAIL	Failure. Unable to parse certificate stream
PS_ARG_FAIL	Failure. Bad input parameters

Parses PEM format certificate file(s) into a psX509Cert_t structure(s). If the fileName parameter contains a list of filenames separated by a semicolon, the loaded certs will be stored in a linked list with head in the next member of outcert. If the file contained multiple certificates, outcert will point to the first certificate, and the next members form a linked list. Note that MatrixSSL APIs typically expect the linked list to be in child-to-parent order. This means that chain certificates should be concatenated in child-to-parent order to the PEM file.

After a successful call, the certificate public key will be accessible in the publicKey member of the psX509Cert_t structure. The algorithm-specific public key (of type psRsaKey_t or psEccKey_t) is stored in the rsa and ecc members of publicKey.

Caller must free outcert with psX509FreeCert on success.

To parse ASN.1 DER format certificates without the PEM wrapping, psX509ParseCert should be used instead of this function.

2.1.3 psX509ParseCert

int32 psX509ParseCert(psPool_t *pool, const unsigned char *pp, uint32 size,
psX509Cert_t **outcert, int32 flags)

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the output certificate will be allocated
pp	input	Pointer to a memory buffer containing an ASN.1 DER encoded X.509 certificate.
outcert	output	The output psX509Cert_t structure. Must be always freed by caller, even on failure.
flags	input	Additional flags. CERT_STORE_UNPARSED_BUFFER: Keep the raw, ASN.1 DER encoded certificate in the unparsedBin member of the psX509Cert structure. CERT_STORE_DN_BUFFER: Store DN Attributes in the psX509Cert structure.

Return Value	Description
>0	Success. A valid psX509Cert structure is allocated and populated in "outcert" parameter. The return value is the size of the parsed certificate.
PS_MEM_FAIL	Failure. Unable to allocate memory for the structure
PS_PARSE_FAIL	Failure. Unable to parse certificate stream
PS_ARG_FAIL	Failure. Bad input parameters

Parses a memory buffer containing one or more ASN.1 DER encoded X.509 certificates into a psX509Cert_t structure.

The certificate public key will be accessible in the publicKey member of the psX509Cert_t structure as a psPubKey_t data type for use in public key APIs. The algorithm-specific public key (psRsaKey_t or psEccKey_t) can be accessed via the rsa and ecc members of the psPubKey_t structure.

Caller must always free outcert with psX509FreeCert, even on failure.

2.1.4 psX509FreeCert

void psX509FreeCert(psX509Cert_t *cert)

Parameter	Input/Output	Description
cert	input	The certificate structure to free

Free a certificate structure allocated by psX509ParseCert or psX509ParseCertFile.

2.1.5 matrixValidateCertsExt

```
int32 matrixValidateCertsExt(psPool_t *pool,  
psX509Cert_t *subjectCerts, psX509Cert_t *issuerCerts,  
char *expectedName, psX509Cert_t **foundIssuer,  
void *hwCtx, void *poolUserPtr,  
const matrixValidateCertsOptions_t *opts)
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the output certificate will be allocated
subjectCerts	input	Pointer to the subject certificate or certificate chain to authenticate. To validate a certificate chain, subjectCert must point to the child-most certificate of the chain, subjectCert->next must point to its parent certificate, and so on. To validate a single, self-signed certificate, the issuerCerts parameter must be set to NULL.
issuerCerts	input	A linked list of trusted CA certs. The list will be searched for the issuer of the parent-most certificate in subjectCerts. If subjectCerts is a self-signed cert, this argument should be set to NULL.
expectedName	input	The expected Subject or Subject Alternative Name of the child-most certificate in subjectCerts.
foundIssuer	output	Pointer to the certificate in issuerCerts that was found to be the issuer of the parent-most certificate in subjectCerts.
poolUserPtr	input	User-specified memory allocation pointer (or NULL)
opts	input	Pointer to a matrixValidateCertsOptions_t struct. The struct can be used to configure the certificate validation process. This struct is obligatory. If the caller wishes to use default options, the memory occupied by the struct should be set to zero.

Return Value	Description
PS_SUCCESS	Success.
PS_CERT_AUTH_FAIL	Failure. Unable to authenticate the subjectCerts chain.
PS_CERT_AUTH_FAIL_PATH_LEN	Failure. The X.509 path length constraint was exceeded.
PS_UNSUPPORTED_FAIL	Failure. Unsupported certificate format.
PS_ARG_FAIL	Failure. Bad input parameters or public-key verification operation failed.
PS_LIMIT_FAIL	Failure. Internal public-key operation failure.

Values for authStatus member of certificate structure	Description
PS_CERT_AUTH_PASS	The certificate was authenticated fully
PS_CERT_AUTH_FAIL_BC	BasicConstraints failure. The issuing certificate did not have CA permissions to issue certificates
PS_CERT_AUTH_FAIL_DN	DistinguishedName failure. The issuing CA did not match the name that the subject identified as its issuer.
PS_CERT_AUTH_FAIL_REVOKED	A CRL has reported the certificate has been revoked
PS_CERT_AUTH_FAIL_SIG	The public key signature verification operation failed.
PS_CERT_AUTH_FAIL_AUTHKEY	The authorityKeyId extension of the subject cert does not match the subjectKeyId of the issuing certificate.
PS_CERT_AUTH_FAIL_PATH_LEN	The certificate chain is longer than allowed as specified by the pathLen field in the basicConstraints extension.
PS_CERT_AUTH_FAIL_EXTENSION	All the above tests passed but there was a violation of the x.509 extension rules. The authFailReason member can be examined to find the specific extension that failed. If authFailFlags has PS_CERT_AUTH_FAIL_VERIFY_DEPTH_FLAG set, then the max_verify_depth limit specified in the opts struct was exceeded instead.

Validates a certificate or a certificate chain against a list of trusted issuer CAs. The validation process follows the *Certificate Path Validation* procedure of section 6 in RFC 5280. For a list of items checked for each certificate, see the *MatrixSSL Developer Guide*.

This function first validates the provided subject certificate chain (`subjectCerts`) up to the parent-most certificate. Next, the list of trusted issuer certs (`issuerCerts`) is searched for the issuer of that parent-most subject certificate. If found, the parent-most subject certificate is validated against the found issuer. In addition, the subject `commonName` or one of the Subject Alternative Name fields of the end-entity (child-most) certificate in `subjectCerts` will be compared matched against the `expectedName` parameter. A pointer to the found issuer will be returned in the `foundIssuer` parameter.

Some aspects of the certificate validation process, such as the maximum certificate chain validation depth and the field against which `expectedName` should be matched, can be configured via the `matrixValidateCertsOptions_t` struct passed in the `opts` parameter. For a description of the available options, please consult the `matrixValidateCertsOptions_t` struct definition in `matrixssllib.h`. Also, the session options section of the *MatrixSSL APIs* reference guide contains some information.

If the subject certificate chain was successfully validated against the supplied issuer list, `PS_SUCCESS` will be returned and every certificate in the `subjectCerts` chain will have its `authStatus` field set to `PS_CERT_AUTH_PASS`.

If any part of the validation process fails, an error code will be returned. More information about the failure will be stored in the `authStatus` field of the `psX509Cert_t` whose validation produced the first failure. The possible values for `authStatus` are listed in the above table.

Note that MatrixSSL checks the certificate validity date during parsing. By default, `matrixValidateCertsExt` will not re-perform date validation. This can be a problem for long-living processes that may outlast the certificate validity period. In this case, it may be a good idea to pass `VCERTS_FLAG_REVALIDATE_DATES` in the flag member of the `opts` argument. This causes `matrixValidateCertsExt` to re-perform date validation independently.

2.1.6 matrixValidateCerts

```
int32 matrixValidateCerts(psPool_t *pool, psX509Cert_t *subjectCerts,
                        psX509Cert_t *issuerCerts, char *expectedName,
                        psX509Cert_t **foundIssuer, void *hwCtx,
                        void *poolUserPtr)
```

A deprecated version of `matrixValidateCertsExt`. This version takes in no optional parameters and attempts to use the default options. New applications should call `matrixValidateCertsExt` instead.

2.1.7 psBase64EncodeAndWrite

```
int32_t psBase64EncodeAndWrite(psPool_t *pool, const char *fileName,
                              unsigned char *bin, uint32_t binLen, int32 fileType,
                              char *hexCipherIV, uint16_t hexCipherIVLen);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
fileName	input	The name of the PEM file where to write the result
bin	input	The data to write to a PEM file
binLen	input	Length of the data.
fileType	input	Type of the file to write. This will affect the PEM encoding. Possible options are REQ_FILE_TYPE (for CSRs), CERT_FILE_TYPE (for certificates), RSA_KEY_FILE_TYPE (RSA keys), ECKEY_FILE_TYPE (ECC keys).
hexCipherIV	input	When writing private key files with 3DES, the IV to use should be passed in in this parameter. This parameter is optional and should be set to NULL except in the above use case.
hexCipherIVLen	input	Length of the hexCipherIV.

Return Value	Description
PS_SUCCESS	Success.
PS_FAILURE	Failure. Error writing the PEM file.

Write an in-memory CSR, certificate or private key to a PEM file.

2.1.8 X.509 Getter API

The X.509 Getter API is intended for accessing parsed data from X.509 certificates and CSRs that it difficult to access directly via the struct member variables.

2.1.8.1 psX509GetCertPublicKeyDer

```
PSPUBLIC int32 psX509GetCertPublicKeyDer(psX509Cert_t *cert,
                                         unsigned char *der_out,
                                         uint16_t *der_out_len);
```

Parameter	Input/Output	Description
cert	input	A psX509Cert_t returned by a successful call to psX509ParseCert or psX509ParseCertFile with the CERT_STORE_UNPARSED_BUFFER flag set.
der_out	output	Pointer to a memory buffer where the extracted public key will be stored.
der_out_len	input/output	Input: size of the der_out buffer. Output: size of the extracted public key.

Return Value	Description
PS_SUCCESS	Success. The public key was extracted successfully.
PS_ARG_FAIL	Failure. Invalid arguments provided: some of the pointer arguments are NULL or cert does not contain usable data from which to extract the public key. The latter could be because the CERT_STORE_UNPARSED_BUFFER flag was not set when calling the certificate parse function.
PS_OUTPUT_LENGTH	Output length negotiation. Size of the output buffer is too small. Please try again with a buffer of size at least the value returned in der_out_len.

Extracts the public key (the SubjectPublicKeyInfo ASN.1 structure) from a parsed X.509 certificate. The public key is returned in DER-encoded format. The certificate must have been parsed with either psX509ParseCert or psX509ParseCertFile and the CERT_STORE_UNPARSED_BUFFER flag must have been set in that call.

Example:

```

#include "matrixssl/matrixsslApi.h"
psX509Cert_t *cert;
unsigned char pubkey_der[4096];
uint16_t pubkey_der_len = sizeof(pubkey_der);
int32 rc;
psCryptoOpen(PS_CRYPTO_CONFIG);
rc = psX509ParseCertFile(NULL, "mycert.pem", &cert,
    CERT_STORE_UNPARSED_BUFFER);
if (rc == PS_SUCCESS) {
    rc = psX509GetCertPublicKeyDer(cert, pubkey_der, &pubkey_der_len);
    if (rc == PS_SUCCESS) {
        printf("Successfully extracted a public key of length %hu\n",
            pubkey_der_len);
    }
}
psX509FreeCert(cert);

```

2.1.8.2 psX509GetOnlineDN

```
int32_t psX509GetOnlineDN(const x509DNAttributes_t *DN,
                          char **out_str,
                          size_t *out_str_len);
```

Parameter	Input/Output	Description
DN	input	The DN struct from which to build the online string.
out_str	output	Pointer to the memory address where the resulting string will be allocated and copied to. Caller is responsible for freeing.
out_str_len	output	Length of the resulting string

Create a concatenated string containing all the supported fields of a DN component.

This function creates a online string from a DN, using a format similar to OpenSSL's X509_NAME_oneline(). Only the fields supported by the current configuration (cryptoConfig.h) are printed. Example output: "C=US, ST=State, DC=com, DC=insidesecure, DC=test/street=street/title=Dr, GN=GivenName, SN=Surname/name=GivenName Surname."

2.1.8.3 psX509GetNumOrganizationalUnits

```
int32_t psX509GetNumOrganizationalUnits(const x509DNAttributes_t *DN);
```

This function returns the number of organizational units in a parsed Distinguished Name struct.

2.1.8.4 psX509GetOrganizationalUnit

```
x509OrgUnit_t *psX509GetOrganizationalUnit(
    const x509DNAttributes_t *DN,
    int32_t index);
```

Parameter	Input/Output	Description
DN	input	Pointer to the filled-in x509DNAttributes_t struct. Note that freeing the DN will invalidate the returned x509OrgUnit_t.
index	input	The index of the organizationalUnit in the order they appear in the DER encoding.

MatrixSSL supports multiple organizationalUnit fields in a Distinguished Name. These are stored in a linked list during parsing. It is possible to access the organizationalUnits in the order they were found in the DER encoded Distinguished Name by using this function.

Caller must NOT free the returned x509OrgUnit_t.

2.1.8.5 psX509GetNumDomainComponents

```
int32_t psX509GetNumDomainComponents(const x509DNAttributes_t *DN);
```


Returns the number of domain components in a parsed Distinguished Name struct.

2.1.8.6 psX509GetDomainComponent

```
x509DomainComponent_t *psX509GetDomainComponent(  
    const x509DNAttributes_t *DN,  
    int32_t index);
```

Parameter	Input/Output	Description
DN	input	Pointer to the filled-in x509DNAttributes_t struct. Note that freeing the DN will invalidate the returned x509DomainComponent_t.
index	input	The index of the domainComponent_t in the order they appear in the DER encoding.

MatrixSSL supports multiple domainComponent fields in a Distinguished Name. These are stored in a linked list during parsing. It is possible to access the domainComponents in the order they were found in the DER encoded Distinguished Name by using this function.

Caller must NOT free the returned x509DomainComponent_t.

2.1.8.7 psX509GetConcatenatedDomainComponent

```
int32_t psX509GetConcatenatedDomainComponent(  
    const x509DNAttributes_t *DN,  
    char **out_str,  
    size_t *out_str_len);
```

Parameter	Input/Output	Description
DN	input	DN The DN struct from which to fetch the domainComponents.
out_str	input	The concatenated domainComponents as a string. This function will malloc a string of suitable length. The caller is responsible for freeing it.
out_str_len	output	Length of the returned string.

Get the concatenation of all domainComponents in a DN as a C string. This function returns the concatenated domainComponents as a string terminated with DN_NUM_TERMINATING_NULLS NULL characters. The output string will contain the components in the reverse order compared to the order in which they were encoded in the certificate. Usually, this will result in the usual print order, i.e. top-level component (.com, .org, ...) last.

2.1.9 X.509 Generation API

For generating certificate signing requests (CSRs) and certificates with MatrixSSL, the user has the option of either using the provided command-line tools in apps/crypto, or an API. For information on how to use the command-line tools, please consult the *Matrix Key and Certificate Generation Utilities* manual. The API for CSR and certificate generation is described in this section.

To enable compilation of the X.509 Generation API, the macro `USE_CERT_GEN` must be defined in `cryptoConfig.h`.

2.1.9.1 Overview

X.509 Generation API is based on the `psCertConfig_t` data structure. This structure contains all the needed information for generating a CSR or a certificate, such as the subject distinguished name (DN) and the certificate extensions. The API user fills in this structure using a setter API. For simple fields, the value is given directly to the setter function as an argument. For complex extensions, a separate structure such as `subjectAltNameEntry_t` is filled-in by the caller and passed to the setter function.

After the `psCertConfig_t` has been constructed using the setter API, it can be passed onwards to the actual CSR and certificate generation functions which use the information contained in it to create the CSR or certificate.

2.1.9.2 Initializing

Before starting to construct a `psCertConfig_t` structure instance with the setter API, the corresponding memory **must** be set to 0:

```
psCertConfig_t conf;
memset(&conf, 0, sizeof(psCertConfig_t));
```

2.1.9.3 Memory allocation

When necessary, the setter function allocate space for new data in the `psCertConfig_t`. Therefore, the API user does not have to do any memory allocation. However, the user **must** call `psX509FreeCertConfig` to free the memory when the `psCertConfig_t` is not used anymore.

2.1.9.4 psX509SetDNAttribute

```
int32 psX509SetDNAttribute(psPool_t *pool,
                           psCertConfig_t *config,
                           const char *name, size_t name_len,
                           const char *value, size_t value_len,
                           int encoding);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
config	input/output	The psCertConfig_t into which the new field value will be added.
name	input	Name of the DN attribute, e.g. "country", "domainComponent", "organization", "organizationalUnit". For a full list of available attribute names, consult x509.h.
name_len	input	The length of the name string.
value	input	The value to be set in the DN attribute. For example, if the name of attribute was specified as "country", a valid value would be "US".
value_len	input	The length of the value string.
encoding	input	ASN.1 encoding to use for the DN value. Only ASN_UTF8STRING is supported. This encoding is recommended in RFC 5280.

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_UNSUPPORTED_FAIL	Failure. Unsupported encoding. Only ASN_UTF8STRING is supported.

This function can be used to set subject DN attributes in a `psCertConfig_t`. The function must be given a name-value pair, where *name* is the name of the DN attribute and *value* is the value to be assigned to that attribute in the `psCertConfig_t`. Note that the characters =, ; and " (equals sign, semicolon and a quote) are not allowed in the value argument.

Example:

```
rc = psX509SetDNAttribute(NULL, conf, "country", strlen("country"),
                          "US", strlen("US"), ASN_UTF8STRING);
if (rc < 0) {
    printf("psX509SetDNAttribute failed\n");
    return PS_FAILURE;
}
```

2.1.9.5 psX509SetValidDays

```
int32 psX509SetValidDays(psPool_t *pool,
                        psCertConfig_t *config,
                        int32 validDays);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
config	input/output	The psCertConfig_t in which the value will be set.
validDays	input	Number of days to use in the validity period. For example, 30 or 365.

Return Value	Description
PS_SUCCESS	Success.

Set the certificate or CSR validity period as a number from days from the current (creation) date.

It is not allowed to call this function after calling `psX509SetValidNotBefore` and `psX509SetValidNotAfter` APIs for the same config struct.

2.1.9.6 psX509SetValidNotBefore

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
config	input/output	The psCertConfig_t in which the value will be set.
date	input	The notBefore date as an ASCII string representing the value of an ASN.1 UTCTime or GeneralizedTime type. See below for details on the format.
date_len	input	Length of date, excluding any terminating null byte.
encoding	input	The ASN.1 type to use in the encoding. Must be either ASN_UTCTIME or ASN_GENERALIZEDTIME.

Set the starting date of the certificate validity period, before which the certificate is not valid.

The date must be supplied as an ASCII string representing the value of an ASN.1 type UTCTime or GeneralizedTime. The ASN.1 type to use must be specified in the `encoding` parameter, which must be either `ASN_UTCTIME` or `ASN_GENERALIZEDTIME`.

The date value must conform to RFC 5280. UTCTime should be used for dates through 2049 and GeneralizedTime should be used for dates in 2050 or later. The accepted format is `YYMMDDHHMMSSZ` for UTCTime and `YYYYMMDDHHMMSSZ` for GeneralizedTime. The time zone must be GMT (also called Zulu time). The date must include exactly two seconds and fractional seconds are not allowed.

It is not allowed to call this function after calling `psX509SetValidDays` for the same config struct.

2.1.9.7 psX509SetValidNotAfter

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
config	input/output	The psCertConfig_t in which the value will be set.
date	input	The notAfter date as an ASCII string representing the value of an ASN.1 UTCTime or GeneralizedTime type. See below for details on the format.
date_len	input	Length of date, excluding any terminating null byte.
encoding	input	The ASN.1 type to use in the encoding. Must be either <code>ASN_UTCTIME</code> or <code>ASN_GENERALIZEDTIME</code> .

Set the end date of the certificate validity period, after which the certificate is no longer valid.

The date must be supplied as an ASCII string representing the value of an ASN.1 type UTCTime or GeneralizedTime. The ASN.1 type to use must be specified in the `encoding` parameter, which must be either `ASN_UTCTIME` or `ASN_GENERALIZEDTIME`.

The date value must conform to RFC 5280. UTCTime should be used for dates through 2049 and GeneralizedTime should be used for dates in 2050 or later. The accepted format is `YYMMDDHHMMSSZ` for UTCTime and `YYYYMMDDHHMMSSZ` for GeneralizedTime. The time zone must be GMT (also called Zulu time). The date must include exactly two seconds and fractional seconds are not allowed.

To set an indefinite expiration date, the special date string 99991231235959Z can be used, with the encoding parameter set to ASN_GENERALIZEDTIME.

It is not allowed to call this function after calling psX509SetValidDays for the same config struct.

2.1.9.8 psX509SetSerialNum

```
int32 psX509SetSerialNum(psPool_t *pool,  
                        psCertConfig_t *config,  
                        unsigned char *serialNum,  
                        size_t serialNumLen);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
config	input/output	The psCertConfig_t in which the value will be set.
serialNum	input	Byte array containing the serial number in binary.
serialNumLen	input	Length of the serialNum array in bytes.

Return Value	Description
PS_SUCCESS	Success.
PS_ARG_FAIL	Failure. The serialNum array is empty.

Set the serial number in the psCertConfig_t.

2.1.9.9 psX509SetCertHashAlg

```
int32 psX509SetCertHashAlg(psPool_t *pool,  
                          psCertConfig_t *config,  
                          int32 certAlg);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
config	input/output	The psCertConfig_t in which the value will be set.
certAlg	input	The certificate hash algorithm to use together with the public key as the signature algorithm.

Return Value	Description
PS_SUCCESS	Success.
PS_UNSUPPORTED_FAIL	Failure. Unsupported hash algorithm.

Sets the hash algorithm that should be used in certificate signature generation. The supported hash algorithm identifiers are: ALG_MD5, ALG_SHA1, ALG_SHA256, ALG_SHA384, ALG_SHA512.

Note that this function only sets the hash algorithm part of the signature algorithm. The actual signature algorithm depends on the public key used to sign it. For example, if the CA generating the certificate uses

an RSA key pair, and the hash algorithm has been set as ALG_SHA256, the certificate signature algorithm will be RSA-SHA256.

2.1.9.10 psX509AddSubjectAltName

```
int32 psX509SetSubjectAddName(psPool_t *pool,  
                               psCertConfig_t *config,  
                               subjectAltNameEntry_t *entry);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
config	input/output	The psCertConfig_t in which the value will be set.
entry	input	A filled-in subjectAltNameEntry_t struct. The subjectAltName extension in psCertConfig_t will be populated with data from this struct.

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_FAILURE	Failure. The data in the subjectAltNameEntry_t struct is invalid.

Add information to the subjectAltName extension in the psCertConfig_t. The information is provided to this function in a filled-in subjectAltNameEntry_t struct, defined in x509.h. The function can be called multiple times to add more information to the extension. However, the subjectAltNameEntry_t struct should be zeroed between the two calls.

Filling the subjectAltNameEntry_t struct is rather straightforward. Some notes are necessary, however.

The ipAddress field must consist of a v4 IP address as a dot-separated string, such "127.2.3.4".

The otherName field can be used for adding extra, user-specified name fields, as specified by RFC 5280.

An otherName contains an OID and a user-specific identifier string. There are two ways to encode the OID in otherName: either as a dot-notation string or as a hex string containing the DER encoding of the OID.

When encoding the OID using dot-notation (recommended), the user must fill in the field

otherNameDotNotation using the format "[dot-notation-OID]:[ASCII-string]", such as

"1.2.840.113549:some other identifier". The length of the otherNameDotNotation string must be

assigned to otherNameDotNotationLen. When encoding the OID as a hex string, the user must fill-in

the otherName field using the format "[hex-encoded-OID]:[ASCII-string]", such as "2ab00f:some other identifier"; The length of the string in otherName must be assigned to otherNameLen.

For information on the meaning of subjectAltName extension fields, please consult RFC 5280.

Example:

```
subjectAltNameEntry_t sanEntry;  
int32 rc;  
  
memset(&sanEntry, 0, sizeof(subjectAltNameEntry_t));  
sanEntry.rfc822Name = "email@address.com";  
sanEntry.rfc822NameLen = strlen(sanEntry.rfc822Name);  
sanEntry.dNSName = "insidesecure.com";  
sanEntry.dNSNameLen = strlen(sanEntry.dNSName);
```

```

sanEntry.uniformResourceIdentifier = "http://www.insidesecure.com";
sanEntry.uniformResourceIdentifierLen = strlen(sanEntry.uniformResourceIdentifier);
sanEntry.iPAddress = "46.105.106.144";
sanEntry.iPAddressLen = strlen(sanEntry.iPAddress);
sanEntry.otherName = "2ab00f:some other identifier";
sanEntry.otherNameLen = strlen(sanEntry.otherName);

rc = psX509SetSubjectAltName(NULL,
                             conf,
                             &sanEntry);
if (rc < 0) {
    printf("psX509SetSubjectAltName failed\n");
    return PS_FAILURE;
}

```

2.1.9.11 psX509AddIssuerAltName

```

int32 psX509SetIssuerAltName(psPool_t *pool,
                             psCertConfig_t *config,
                             subjectAltNameEntry_t *entry);

```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
config	input/output	The psCertConfig_t in which the value will be set.
entry	input	A filled-in subjectAltNameEntry_t struct. The issuerAltName extension in psCertConfig_t will be populated with data from this struct.

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_FAILURE	Failure. The data in the subjectAltNameEntry_t struct is invalid.

Add information to the issuerAltName extension in the psCertConfig_t. The issuerAltName extension is almost identical to the subjectAltName extension. The operation of this function is identical to psX509AddSubjectAltName.

2.1.9.12 psX509AddKeyUsageBit

```

int32 psX509AddKeyUsageBit(psPool_t *pool,
                             psCertConfig_t *config,
                             const char *usageBitName);

```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
config	input/output	The psCertConfig_t in which the value will be set.
usageBitName	input	Name of the keyUsage bit to set.

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_ARG_FAIL	Failure. The usageBitName is illegal.

Set a keyUsage bit in the psCertConfig_t. The supported keyUsage bits are: keyCertSign, keyAgreement, crlSign, digitalSignature, keyEncipherment, dataEncipherment, nonrepudiation, encipherOnly, decipherOnly. For details on the keyUsage extension and the meaning of the bits, see RFC 5280. The user is responsible for ensuring that the keyUsage bit combination is sensible. For example, RFC 5280 states that encipherOnly and decipherOnly should only be used together with keyAgreement.

This function can be called multiple times to add different bits. The bits can be cleared with psX509ClearKeyBitUsageBits.

2.1.9.13 psX509AddExtendedKeyUsage

```
int32 psX509AddExtendedKeyUsage(psPool_t *pool,
                                psCertConfig_t *config,
                                const char *usage);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
config	input/output	The psCertConfig_t in which the value will be set.
usage	input	Name of the extendedKeyUsage bit to set.

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_ARG_FAIL	Failure. The function arguments are invalid or the extendedKeyUsage bit is illegal.

Set keyUsage bits in the extendedKeyUsage extension. The supported bits are serverAuth, clientAuth and codeSigning.

2.1.9.14 psX509SetSubjectKeyId

```
int32 psX509SetSubjectKeyId(psPool_t *pool,
                             psCertConfig_t *config,
                             unsigned char *id,
                             size_t len);
```


Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
config	input/output	The psCertConfig_t in which the value will be set.
usage	input	The value of the subjectKeyID, provided in binary as a byte array.
len	input	Length of the usage array.

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_ARG_FAIL	Failure. The function arguments are invalid or the provided subjectKeyID is too long.

Set the subjectKeyIdentifier extension in the psCertConfig_t.

2.1.9.15 psX509SetBasicConstraintsCA

```
int32 psX509SetBasicConstraintsCA(psPool_t *pool,
                                   psCertConfig_t *config,
                                   int caBit);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
config	input/output	The psCertConfig_t in which the value will be set.
caBit	input	The value of the CA bit. 1==true and 0==false.

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_ARG_FAIL	Failure. The function arguments are invalid.

Set the CA bit in the basicConstraints extension in the psCertConfig_t.

2.1.9.16 psX509SetBasicConstraintsPathLen

```
int32 psX509SetBasicConstraintsPathLen(psPool_t *pool,
                                         psCertConfig_t *config,
                                         int pathLenConstraint);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
config	input/output	The psCertConfig_t in which the value will be set.
pathLenConstraint	input	The value of the path length constraint.

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_ARG_FAIL	Failure. The function arguments are invalid.

Set the value of the path length constraint in the basicConstraints extension in the psCertConfig_t.

2.1.9.17 psX509AddAuthorityInfoAccess

```
int32 psX509AddAuthorityInfoAccess(psPool_t *pool,
                                   psCertConfig_t *config,
                                   authorityInfoAccessEntry_t *entry);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
config	input/output	The psCertConfig_t in which the value will be set.
entry	input	A filled-in authorityInfoAccessEntry_t struct. The authorityInfoAccess extension in psCertConfig_t will be populated with data from this struct.

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_FAILURE	Failure. The data in the authorityInfoAccessEntry_t struct is invalid.

Add an entry to the authorityInfoAccess extension in the psCertConfig_t. As in psX509SetSubjectAltName, the extension data is provided to this function via a fill-in struct (authorityInfoAccessEntry_t). The function can be called multiple times to add more entries into the extension. Both ocsp and calssuers can be set in a single call, although these are separate AccessDescription entries in the authorityInfoAccess ASN.1 definition. Note that accessMethod values will be encoded as uniformResourceIdentifiers.

The following example adds two entries (four AccessDescriptions) via two calls to this function:

```
memset(&authInfo, 0, sizeof(authorityInfoAccessEntry_t));
authInfo.ocsp = "ocsp.insidesecure.com";
authInfo.ocspLen = strlen("ocsp.insidesecure.com");
authInfo.calssuers = "ca.insidesecure.com";
authInfo.calssuersLen = strlen("ca.insidesecure.com");

rc = psX509AddAuthorityInfoAccess(NULL,
                                   &conf,
                                   &authInfo);

if (rc < 0) {
    printf("psX509AddAuthorityInfoAccess failed\n");
    return PS_FAILURE;
}

memset(&authInfo, 0, sizeof(authorityInfoAccessEntry_t));
```

```

authInfo.ocsp = "ocsp2.insidesecure.com";
authInfo.ocspLen = strlen("ocsp2.insidesecure.com");
authInfo.caIssuers = "ca2.insidesecure.com";
authInfo.caIssuersLen = strlen("ca2.insidesecure.com");

rc = psX509AddAuthorityInfoAccess(NULL,
                                   &conf,
                                   &authInfo);

if (rc < 0) {
    printf("psX509AddAuthorityInfoAccess failed\n");
    return PS_FAILURE;
}

```

2.1.9.18 psX509AddPolicy

```

int32 psX509AddPolicy(psPool_t *pool,
                      psCertConfig_t *config,
                      certificatePoliciesEntry_t *entry);

```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
config	input/output	The psCertConfig_t in which the value will be set.
entry	input	A filled-in certificatePoliciesEntry_t struct. The certificatePolicies extension in psCertConfig_t will be populated with data from this struct.

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_FAILURE	Failure. The data in the certificatePoliciesEntry_t struct is invalid.

Add a policy to the certificatePolicies extension in the psCertConfig_t struct. As in psX509SetSubjectAltName, the extension data is provided to this function via a fill-in struct (certificatePoliciesEntry_t). The function can be called multiple times to add more entries into the extension.

The certificatePolicies extension is a hierarchical structure. Every policy (PolicyInformation ASN.1 type) can contain multiple PolicyQualifierInfos, which, in turn, can contain multiple UserNotices. To specify which policy to modify, the policyIndex member in the certificatePoliciesEntry_t structure can be used. Similarly, to specify which UserNotice within the policy to modify, the unoticeIndex field can be used. The indexing **must** start from 1. In other words, the first policy must have index 1 and the first UserNotice within a policy must also have index 1.

Note that the policy OID must be given as a hex string containing the DER-encoding of the OID.

In the following example, a policy with a CPS and a UserNotice is added in the first function call. In the second call, another UserNotice is added to the same policy. In the third call, a second policy with a CPS is added.

```

/* Add a policy with OID. */
memset(&pols, 0, sizeof(certificatePoliciesEntry_t));
pols.policyIndex = 1; /* Note: indexing MUST start from 1. */

```

```

pols.policyOid = "67810C010201";
pols.policyOidLen = strlen(pols.policyOid);

/* Add CPS to policy #1 */
pols.cps = "http://www.insidesecond.com/cps1";
pols.cpsLen = strlen(pols.cps);

/* Add UserNotice to policy #1 */
pols.unoticeIndex = 1; /* Note: indexing MUST start from 1. */
pols.unoticeOrganization = "INSIDE Secure Oyj 1";
pols.unoticeOrganizationLen = strlen(pols.unoticeOrganization);
pols.unoticeExplicitText = "Explicit Text 1";
pols.unoticeExplicitTextLen = strlen(pols.unoticeExplicitText);
pols.unoticeNumbers[0] = 1;
pols.unoticeNumbers[1] = 2;
pols.unoticeNumbersLen = 2;

rc = psX509AddPolicy(NULL,
                    &conf,
                    &pols);

if (rc < 0) {
    printf("psX509AddPolicy failed\n");
    return PS_FAILURE;
}

/* Add another UserNotice to policy #1. */
memset(&pols, 0, sizeof(certificatePoliciesEntry_t));
pols.policyIndex = 1;
pols.unoticeIndex = 2;
pols.unoticeOrganization = "INSIDE Secure Oyj 2";
pols.unoticeOrganizationLen = strlen(pols.unoticeOrganization);
pols.unoticeExplicitText = "Explicit Text 2";
pols.unoticeExplicitTextLen = strlen(pols.unoticeExplicitText);
pols.unoticeNumbers[0] = 3;
pols.unoticeNumbers[1] = 4;
pols.unoticeNumbersLen = 2;

rc = psX509AddPolicy(NULL,
                    &conf,
                    &pols);

if (rc < 0) {
    printf("psX509AddPolicy failed\n");
    return PS_FAILURE;
}

```

```

/* Now add policy #2 with a different OID. */
memset(&pols, 0, sizeof(certificatePoliciesEntry_t));
pols.policyIndex = 2;
pols.policyOid = "67810C010202";
pols.policyOidLen = strlen(pols.policyOid);

/* Add CPS to policy #2 */
pols.cps = "http://www.insidesecure.com/cps2";
pols.cpsLen = strlen(pols.cps);

rc = psX509AddPolicy(NULL,
                     &conf,
                     &pols);

if (rc < 0) {
    printf("psX509AddPolicy failed\n");
    return PS_FAILURE;
}

```

2.1.9.19 psX509SetConstraintRequireExplicitPolicy

```

psX509SetConstraintRequireExplicitPolicy(psPool_t *pool,
                                         psCertConfig_t *config,
                                         int32_t value);

```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
config	input/output	The psCertConfig_t in which the value will be set.
value	input	The value of the requireExplicitPolicy constraint.

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_ARG_FAIL	Failure. The function arguments are invalid.

Add a requireExplicitPolicy constraint with a given value to the policyConstraints extension in the psCertConfig_t. For a description of the policyConstraints extension, see RFC 5280.

2.1.9.20 psX509SetConstraintInhibitPolicyMappings

```

int32 psX509SetConstraintInhibitPolicyMappings(psPool_t *pool,
                                              psCertConfig_t *config,
                                              int32_t value);

```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
config	input/output	The psCertConfig_t in which the value will be set.
value	input	The value of the inhibitPolicyMappings constraint.

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_ARG_FAIL	Failure. The function arguments are invalid.

Add a inhibitPolicyMappings constraint with a given value to the policyConstraints extension in the psCertConfig_t. For a description of the policyConstraints extension, see RFC 5280.

2.1.9.21 psX509AddPolicyMapping

```
int32 psX509AddPolicyMapping(psPool_t *pool,
                             psCertConfig_t *config,
                             char *issuerDomainPolicy,
                             size_t issuerDomainPolicyLen,
                             char *subjectDomainPolicy,
                             size_t subjectDomainPolicyLen);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
config	input/output	The psCertConfig_t in which the value will be set.
issuerDomainPolicy	input	The value of the issuerDomainPolicy in the mapping. This must be a hex string containing the DER-encoding of the policy OID.
issuerDomainPolicyLen	input	The length of the issuerDomainPolicy string
subjectDomainPolicy	input	The value of the subjectDomainPolicy in the mapping. This must be a hex string containing the DER-encoding of the policy OID.
subjectDomainPolicyLen	input	The length of the subjectDomainPolicy string

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_ARG_FAIL	Failure. The function arguments are invalid.
PS_PARSE_FAIL	Failure. The OID hex strings could not be parsed.

Add a policy mapping into the policyMappings extensions in the psCertConfig_t.

2.1.9.22 psX509SetNetscapeComment

```
int32 psX509SetNetscapeComment(psPool_t *pool,
                                psCertConfig_t *certConfig,
                                const char *comment,
                                size_t commentLen);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
config	input/output	The psCertConfig_t in which the value will be set.
comment	input	The value of the netscape-comment string
commentLen	input	Length of the netscape-comment string

Add a netscape-comment extension to the psCertConfig_t.

2.1.9.23 psX509SetPublicKey

```
int32 psX509SetPublicKey(psPool_t *pool,
                        psCertConfig_t *certConfig,
                        psPubKey_t *pk);
```

Set the public key in the psCertConfig_t.

2.1.9.24 psWriteCertReqMem

```
int32 psWriteCertReqMem(psPool_t *pool, psPubKey_t *key,
                        psCertConfig_t *reqConfig,
                        unsigned char **requestMem, int32 *requestMemLen)
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
key	input	The keypair to use with the CSR. The private key in the psPubKey_t will be used to sign the CSR, while the public key in the psPubKey_t will be inserted into the CSR. RSA and ECC keypairs are supported.
reqConfig	input	The psCertConfig_t to use for creating the CSR. This must have been filled using calls to the psCertConfig_t setter API described above.
requestMem	output	A pointer to where the created CSR will be stored.
requestMemLen	output	The length of the created CSR in bytes.

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_FAILURE	Failure. Encoding of the request failed.

Create a certificate signing request (CSR) from a keypair and a psCertConfig_t. The psCertConfig_t structure must have been filled in previously using the setter API described above. Except for the public key, which is provided via the key parameter, the data to be included in the CSR is copied from the provided psCertConfig_t.

This function will generate the CSR in PKCS #10 format using DER encoding. It is possible to write the resulting CSR into a PEM file using psBase64EncodeAndWrite with REQ_FILE_TYPE as the fileType parameter.

This function will allocate memory for the generated CSR. That memory must be freed by the caller when the CSR is no longer used.

2.1.9.25 psParseCertReqFile

```
int32 psParseCertReqFile(psPool_t *pool, const char *fileName,  
                        unsigned char **reqOut, int32 *reqOutLen);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool.
filename	input	Filename of the CSR to parse.
reqOut	output	Pointer to the output buffer where the parsed CSR will be stored.
reqOutLen	output	Length of the parsed CSR.

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_FAILURE	Failure. Parsing of the CSR failed.

Parse a CSR file and stores the parsed CSR in DER encoded form in the provided buffer. The output of this function can be fed to psParseCertReqBuf, which will extract the CSR parts from the DER buffer. This function will allocate enough space for the buffer. The caller is responsible for freeing the memory.

2.1.9.26 psParseCertReqBuf

```
int32 psParseCertReqBuf(psPool_t *pool, unsigned char *reqBuf,  
                      int32 reqBufLen, x509DNattributes_t **DN, psPubKey_t **key,  
                      x509v3extensions_t **reqExt);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool.
reqBuf	input	Pointer to the buffer containing the DER encoded CSR.
reqBufLen	input	Length of the CSR buffer
DN	output	The subject distinguished name (DN) attributes extracted from the CSR.
key	output	The public key extracted from the CSR.
reqExt	output	The extensions extracted from the CSR.

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_FAILURE	Failure. Parsing of the CSR failed.

Parses a DER encoded CSR and extracts the subject DN, the public key and the extensions. These can be further passed onwards to psX509SetCAIssuedCertExtensions.

If the CSR contains the subjectKeyIdentifier extension, this function will ignore it and recreate the subjectKeyIdentifier from the CSR's public key, according to the procedure in section 4.2.1.2 in RFC 5280.

This function will allocate space for the public key, the DN and the extensions. The caller is responsible for freeing the allocated memory. The outputs of this function (DN, key and reqExt) **must** be freed as follows:

DN must be freed with psX509FreeDNStruct followed by psFree

key must be freed with psDeletePubKey
reqExt must be freed with psX509FreeExtensions followed by psFree

For an example on how to use this function, see the documentation for psX509WriteCAIssuedCert.

2.1.9.27 psParseCertReqBufExt

```
extern int32 psParseCertReqBufExt(psPool_t *pool, unsigned char *reqBuf,  
                                int32 reqBufLen, x509DNattributes_t **DN, psPubKey_t **key,  
                                x509v3extensions_t **reqConfig, psCertReq_t **parsedReq);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool.
reqBuf	input	Pointer to the buffer containing the DER encoded CSR.
reqBufLen	input	Length of the CSR buffer
DN	output	The subject distinguished name (DN) attributes extracted from the CSR.
key	output	The public key extracted from the CSR.
reqExt	output	The extensions extracted from the CSR.
parsedReq	output	A struct containing additional information parsed from the CSR.

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_FAILURE	Failure. Parsing of the CSR failed.

This function is similar to psParseCertReqBuf, except that it stores some additional information about the certificate signing request into a psCertReq_t struct. See the definition of that struct in x509.h for a list of stored items. The items can be accessed from the psCertReq_t struct via the psCertReqGet* set of functions.

2.1.9.28 psCertReqGetSignatureAlgorithm

```
int32_t psCertReqGetSignatureAlgorithm(psCertReq_t *req);
```

This function returns the signature algorithm that was used to sign the parsed CSR. More specifically, this function returns the value of the signatureAlgorithm field (in MatrixSSL's OID format) of the CertificationRequest ASN.1 structure contained in the parsed CSR. The psCertReq_t struct should have been previously filled in by psParseCertReqBufExt.

The return value is an integer representing the algorithm OID in MatrixSSL format, for example OID_SHA256_RSA_SIG or OID_SHA256_ECDSA_SIG. For a full list of MatrixSSL format OID definitions, consult cryptolib.h.

2.1.9.29 psCertReqGetPubKeyAlgorithm

```
int32_t psCertReqGetPubKeyAlgorithm(psCertReq_t *req);
```

This function returns the public key algorithm field of the parsed CSR. More specifically, this function returns the `AlgorithmIdentifier` field (in MatrixSSL's OID format) of the `SubjectPublicKey` ASN.1 structure contained in the `CertificateRequestInfo` structure of the parsed CSR. The `psCertReq_t` struct should have been previously filled in by `psParseCertReqBufExt`.

The return value is either `OID_RSA_KEY_ALG` or `OID_ECDSA_KEY_ALG`, as those are the supported public key algorithms supported by MatrixSSL in CSR parsing.

2.1.9.30 `psCertReqGetVersion`

```
int32_t psCertReqGetVersion(psCertReq_t *req)
```

This function returns the version number of the parsed CSR. More specifically, this function returns the `version` field of the `CertificateRequestInfo` ASN.1 structure of the parsed CSR. The `psCertReq_t` struct should have been filled in by `psParseCertReqBufExt`.

2.1.9.31 `psX509SetCAIssuedCertExtensions`

```
int32 psX509SetCAIssuedCertExtensions(psPool_t *pool,
                                       psCertConfig_t *certConfig,
                                       x509v3extensions_t *reqExt,
                                       psX509Cert_t *caX509);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the <code>psCertConfig_t</code> members will be allocated
config	input/output	Input: the CA's configuration structure containing e.g. the CA's distinguished name and extensions such as <code>subjectKeyId</code> and <code>authorityInfoAccess</code> . Output: a modified <code>psCertConfig_t</code> that can be passed to <code>psX509WriteCAIssuedCert</code> .
reqExt	input	The <code>x509v3extensions</code> structure from the certificate signing request (CSR) from which the CA-issued certificate is to be created.
caX509	input	The CA's certificate.

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_ARG_FAIL	Failure. The function arguments are invalid.

This function is used to setup the extensions in a `psCertConfig_t` for the purpose of creating a CA-signed certificate. The extensions are extracted from different from three places: the certificate signing request (CSR), the CA's configuration (`psCertConfig_t`) and (for the `subjectKeyId` extension only) from the CA's certificate. The final set of extensions is stored in the `psCertConfig_t` pointed to by the `certConfig` parameter.

For the `basicConstraints`, `subjectAltName`, `keyUsage` and `extendedKeyUsage` extensions, if both the CSR and the CA's config structure contain one of these extensions, the *one in the CSR* will override the one in the CA's config.

This function will set the `authorityKeyId` extension in `certConfig` to the same value as the `subjectKeyId` in the CA's config struct.

Note that this function **must** be called for a `psCertConfig_t` before it is passed to the `psX509WriteCAIssuedCert` function.

2.1.9.32 psX509WriteCAIssuedCert

```
int32_t psX509WriteCAIssuedCert(psPool_t *pool, psCertConfig_t *certConfig,  
                                psPubKey_t *reqPubKey, char *subjectDN, int32_t subjectDNLen,  
                                psX509Cert_t *caCert, psPubKey_t *caPrivKey,  
                                char *certFileOut);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
certConfig	input	The configuration file containing the extensions to use for the generated certificate.
reqPubKey	input	The public key from the CSR.
subjectDN	input	The subject distinguished name of the end-entity requesting the certificate.
subjectDNLen	input	Length of the subjectDN.
caCert	input	The CA's certificate.
caPrivKey	input	The CA's private key.
certFileOut	input	The filename of the output file where the generated certificate will be written.

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_FAILURE	Failure. Consult the error message output for details on what went wrong.

Create a CA-signed certificate from the provided CSR. Additional data must be provided via a psCertConfig_t structure and from a parsed CA certificate. Note that the CA certificate **must** have been parsed with either psX509ParseCert or psX509ParseCertFile with the CERT_STORE_DN_BUFFER flag parameter set.

Example:

```
int32 rc;  
unsigned char *csr;  
int32 csr_len;  
x509DNAttributes_t *subjectDN = NULL;  
psPubKey_t *reqPubKey;  
x509v3extensions_t *reqExt;  
psX509Cert_t *ca_cert;  
psCertConfig_t conf;  
  
rc = psParseCertReqFile(NULL,  
                        csr_filename,  
                        &csr,  
                        &csr_len);  
  
if (rc < 0) {  
    printf("psX509ParseCertReqFile failed\n");  
    return PS_FAILURE;  
}
```

```

rc = psParseCertReqBuf(NULL,
                        csr,
                        csr_len,
                        &subjectDN,
                        &reqPubKey,
                        &reqExt);

if (rc < 0) {
    printf("psX509ParseCertReqBuf failed\n");
    return PS_FAILURE;
}

rc = psX509ParseCertFile(NULL,
                        (char*)ca_cert_filename,
                        &ca_cert,
                        CERT_STORE_DN_BUFFER);

if (rc < 0) {
    printf("psX509ParseCertFile failed\n");
    return PS_FAILURE;
}

```

/* Here, set up conf using the setter API. In this example, we only set validDays and the certificate hash algorithm. */

```

rc = psX509SetValidDays(NULL,
                        &conf,
                        365);

if (rc < 0) {
    printf("psX509SetValidDays failed\n");
    return PS_FAILURE;
}

rc = psX509SetCertHashAlg(NULL,
                        &conf,
                        ALG_SHA256);

if (rc < 0) {
    printf("psX509SetCertHashAlg failed\n");
    return PS_FAILURE;
}

rc = psX509SetCAIssuedCertExtensions(NULL,
                                    &conf,
                                    reqExt,
                                    ca_cert);

if (rc < 0) {
    printf("psX509SetCAIssuedCertExtensions failed\n");
}

```

```

        return PS_FAILURE;
    }

    rc = psX509WriteCAIssuedCert(NULL,
                                  &conf,
                                  reqPubKey,
                                  subjectDN->dnenc,
                                  subjectDN->dnencLen,
                                  ca_cert,
                                  caKeyPair,
                                  (char*)out_filename);

    if (rc < 0) {
        printf("psX509WriteCAIssuedCert failed\n");
        return PS_FAILURE;
    }

    printf("Wrote: %s\n", out_filename);

    psX509FreeCert(ca_cert);
    psX509FreeCertConfig(&conf);
    x509FreeExtensions(reqExt);
    psFree(reqExt, NULL);
    psX509FreeDNStruct(subjectDN, NULL);
    psFree(subjectDN, NULL);
    psDeletePubKey(&reqPubKey);
    psFree(csr, NULL);

```

2.1.9.33 psX509WriteSelfSignedCert

```

int32_t psX509WriteSelfSignedCert(psPool_t *pool,
                                   psCertConfig_t *certConfig,
                                   psPubKey_t *selfSigningPrivKey,
                                   char *certFileOut)

```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the psCertConfig_t members will be allocated
certConfig	input	The configuration file containing the certificate fields, extensions and DN attributes to use for the generated certificate.
selfSigningPrivKey	input	The private key to use for signing the certificate.
certFileOut	input	Filename for the output certificate.

Return Value	Description
PS_SUCCESS	Success.
PS_MEM_FAIL	Failure. Out of memory.
PS_FAILURE	Failure. Consult the error message output for details on what went wrong.

This function will write a self-signed certificate into a PEM-format file. There is also an in-memory version called `psX509WriteSelfSignedCertMem`, which gives out the DER encoding of the created certificate instead of storing it in a PEM file.

2.2 Certificate Revocation List API

These supported functions are implemented as part of the supported crypto package of Matrix distributions and will enable applications to manage certificate revocation lists (CRLs).

2.2.1 `psX509Crl_t` data type

```
typedef struct x509revoked {
    unsigned char    *serial;
    uint16_t         serialLen;
    struct x509revoked *next;
} x509revoked_t;

typedef struct psCRL {
    psPool_t         *pool;
    int32_t           authenticated; /* Has this CRL been authenticated */
    unsigned char     sigHash[MAX_HASH_SIZE];
    int32_t           sigHashLen;
    int32_t           nextUpdateType;
    char             *nextUpdate; /* Only concerned about expiration */
    int32_t           sigAlg;
    unsigned char     *sig;
    uint16_t          sigLen;
    uint16_t          expired;
    x509DNattributes_t issuer;
    x509v3extensions_t extensions;
    x509revoked_t     *revoked;
    struct psCRL      *next;
} psX509Crl_t;
```

2.2.2 `psX509ParseCRL`

```
int32_t psX509ParseCRL(psPool_t *pool, psX509Crl_t **crl,
    unsigned char *crlBin, int32_t crlBinLen);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the output CRL will be allocated
crl	output	The output CRL structure. Must be freed by caller
crlBin	input	The DER formatted CRL stream
crlBinLen	input	Byte length of crlBin

Return Value	Description
PS_SUCCESS	Success. A valid CRL structure is allocated and populated in "crl" parameter
PS_MEM_FAIL	Failure. Unable to allocate memory for the structure
PS_PARSE_FAIL	Failure. Unable to parse CRL stream
PS_ARG_FAIL	Failure. Bad input parameters

Parses a CRL DER stream into a Matrix `psX509Crl_t` structure.

2.2.3 psCRL_Update

```
int psCRL_Update(psX509Crl_t *CRL, int deleteExisting);
```

Parameter	Input/Output	Description
CRL	input	A parsed CRL to be added to the global CRL cache
deleteExisting	input	1 to replace an existing CRL if found. 0 to append

Return Value	Description
0	Failure. The CRL parameter was NULL or already existed in cache
1	Success. CRL was added to cache

Add the CRL reference to the global CRL cache.

IMPORTANT: A direct reference to the supplied pointer is stored in the cache. A copy of the CRL is not added to the global cache. Therefore, if the CRL is deleted in the future via `psCRL_Delete` or `psX509FreeCRL` it will be removed from the cache and the memory freed.

2.2.4 psCRL_determineRevokedStatus

```
int32_t psCRL_determineRevokedStatus(psX509Cert_t *cert);
```

Parameter	Input/Output	Description
cert	input	The cert to test for the revoked status

Return Value	Description
0	Failure. NULL cert parameter
CRL_CHECK_CRL_EXPIRED	Success. CRL is found but has expired. No revoked tests were run on certificate
CRL_CHECK_PASSED_AND_AUTHENTICATED	Success. Authenticated CRL was found for this certificate and the certificate has not been revoked
CRL_CHECK_PASSED_BUT_NOT_AUTHENTICATED	Success. CRL was found for this certificate and the certificate has not been revoked BUT the CRL has not been authenticated
CRL_CHECK_REVOKED_AND_AUTHENTICATED	Success. Authenticated CRL was found for this certificate and the certificate has been revoked
CRL_CHECK_REVOKED_BUT_NOT_AUTHENTICATED	Success. CRL was found for this certificate and the certificate has been revoked BUT the CRL has not been authenticated
CRL_CHECK_EXPECTED	Success. No CRL was found in the global CRL cache but the certificate had a CRL distribution point. The CRL should be fetched
CRL_CHECK_NOT_EXPECTED	Success. No CRL was found in the global CRL cache but this certificate did not have a CRL distribution point so a CRL probably does not exist for this certificate.

Run the given certificate through the revocation tests. The value of `revokedStatus` of the `psX509Cert_t` structure will be set to whatever the return value of this function call is.

2.2.5 psCRL Delete

```
int psCRL_Delete(psX509Crl_t *CRL);
```

Parameter	Input/Output	Description
CRL	input	The CRL to be deleted from the global CRL cache

Return Value	Description
0	Failure. The CRL parameter was NULL or didn't exist
1	Success. CRL was deleted from the cache

Delete a CRL from the global CRL cache and free the memory of the CRL. This function has the same behaviour as `psX509FreeCRL` but was given a `psCRL_` prefix to create a consistent set of functions that manage the global CRL cache.

2.2.6 psCRL DeleteAll

```
void psCRL_DeleteAll();
```

Deletes all CRLs from the global CRL cache and frees the memory for each CRL.

2.2.7 psCRL Remove

```
int psCRL_Remove(psX509Crl_t *CRL);
```

Parameter	Input/Output	Description
CRL	input	The CRL to be removed from the global CRL cache

Return Value	Description
0	Failure. The CRL parameter was NULL or didn't exist
1	Success. CRL was removed from the cache

Remove a CRL from the global CRL cache but do not free the memory of the CRL. The CRL may be deleted with `psX509FreeCRL` at a later time.

2.2.8 psCRL_RemoveAll

```
void psCRL_RemoveAll();
```

Removes all CRLs from the global CRL cache but does not free the memory. Each managed CRLs may be deleted with `psX509FreeCRL` at a later time.

2.2.9 psX509FreeCRL

```
void psX509FreeCRL(psX509Crl_t *crl);
```

Parameter	Input/Output	Description
crl	input	The CRL structure to free

Free a CRL structure allocated by `psX509ParseCRL`. If the CRL entry had been added to the global CRL cache with `psCRL_Update` or `psCRL_Insert` it will be deleted from the cache.

2.2.10 psCRL_GetCRLForCert

```
psX509Crl_t* psCRL_GetCRLForCert(psX509Cert_t *cert);
```

Parameter	Input/Output	Description
cert	input	The cert used to search for the associated CRL

Return Value	Description
NULL	Failure. No matching CRL was found
<valid pointer>	Success. CRL is found and referenced by returned pointer

Locate a CRL for a given certificate in the global CRL cache. This function is useful to locate a CRL from the global cache to be deleted when the certificate is reporting a value of `CRL_CHECK_CRL_EXPIRED` as the `revokedStatus` status.

2.2.11 psX509GetCRLdistURL

```
int32_t psX509GetCRLdistURL(psX509Cert_t *cert, char **url, uint32_t *urlLen);
```

Parameter	Input/Output	Description
cert	input	A parsed certificate from which to search for the CRL distribution point
url	output	A pointer to the URL distribution point or NULL if not found
urlLen	output	Byte length of url

Return Value	Description
PS_TRUE	Success. The url parameter will point to the URL distribution point of the CRL
PS_FALSE	Failure. Certificate did not contain a URL distribution point for a CRL
PS_ARG_FAIL	Failure. Bad input parameters

Return the URL of where to find the CRL for a given certificate. The `url` value will point directly into the read-only `psX509Cert_t` structure and should not be destructively parsed or freed.

A known limitation is that this function will only return the first distribution point that is found in a certificate.

2.2.12 psX509AuthenticateCRL

```
int32_t psX509AuthenticateCRL(psX509Cert_t *CA, psX509Crl_t *CRL,
void *userPtr);
```

Parameter	Input/Output	Description
CA	input	The issuing certificate of the CRL that will be used to authenticate the CRL signature
CRL	input	The CRL to authenticate
userPtr	input	NULL or user context for internal memory pool usage

Return Value	Description
PS_SUCESS	Success. CRL is authenticated and the "authenticated" member has been set to PS_TRUE
PS_CERT_AUTH_FAIL_EXTENSION	Failure. Certificate extensions did not match what the CRL reported as the issuer
PS_CERT_AUTH_FAIL_DN	Failure. Certificate name did not match what the CRL reported as the issuer
PS_UNSUPPORTED_FAIL	Failure. Signature algorithm of CRL is not supported
PS_CERT_AUTH_FAIL_SIG	Failure. Signature authentication failed
PS_MEM_FAIL	Failure. Memory allocation error
PS_ARG_FAIL	Failure. Bad input parameters

Performs the authentication tests on a CRL given an issuer. If the authentication is successful the `authenticated` member of the CRL structure will be set to `PS_TRUE` (1).

This function will always attempt to perform the authentication so the `authenticated` member of the CRL will be reset to 0 at the beginning of this function regardless of the current value.

This function is internally invoked as part of the `psX509AuthenticateCert` logic to handle use cases where a server has presented a certificate chain and the parent certificate will only be available internally during that handshake time window.

2.3 Distinguished Name Attributes

The following table lists the Distinguished Name (DN) attributes supported by MatrixSSL, together with the compilation options needed to enable them.

Compilation option	Distinguished Name Attributes
always enabled	Attributes listed as “MUST support” in RFC 5280: country, organization, organizationalUnit, dnQualifier, serialNumber, state, commonName, domainComponent
#define USE_EXTRA_DN_ATTRIBUTES_RFC5280_SHOULD	Attributes listed as “SHOULD support” in RFC 5280: locality, title, surname, givenName, initials, pseudonym, generationQualifier
#define USE_EXTRA_DN_ATTRIBUTES	Attributes not mentioned in RFC 5280: streetAddress, postalAddress, telephoneNumber, uid, name, email

2.4 Certificate Extensions

2.4.1 Supported Extensions

The following table lists the certificate extensions supported by MatrixSSL during certificate parsing.

Supported Extension	Description
basicConstraints	Identifies a cert as a CA and how long a certificate chain it will allow
nameConstraints	This extension is used in CA certificates to restrict the name space within which all subject names in subsequent certificates must be located.
subjectAltName	Alternative names that associate specific DNS, IP Address, and other identification with the certificate
issuerAltName	Analogous to subjectAltName, but contains issuer information
subjectKeyId	Fingerprint of the subject’s public key (automatically included)
authorityKeyId	Fingerprint of the issuer’s public key (automatically included)
keyUsage	Supports keyAgreement and keyCertSign usage
extendedKeyUsage	Supports TLS Server Authentication and TLS Client Authentication
authorityInfoAccess	This extension is used to specify ways for accessing information about a CA. The most common use is to provide the URI of the CA’s OCSP responder in this extension
certificatePolicies	Policies indicating the terms under which the certificate should be used
policyConstraints	The requireExplicitPolicy and inhibitPolicyMappings constraints are supported
policyMappings	This extension is used in CA certificates to specify pairs of policies that the CA considers to be equivalent
cRLDistributionPoints	Used in CA certificates to specify the locations of the certificate revocation lists (CRLs)

2.4.2 Accessing information in certificate extensions

After a certificate has been successfully parsed, the information contained in the extensions can be accessed in the extensions member of the `psX509Cert_t`. In most cases, the parsed extension contents are stored as C strings, together with the string length. If an extension contains multiple entries of a single type, these entries are parsed into a linked list.

The following example checks whether the parsed certificate contains the authorityInfoAccess extension. If it does, it counts out the number of OCSP responder URI entries in that extension, as well as their total length.

```
#include "matrixssl/matrixsslApi.h"
psX509Cert_t *cert;
psx509authorityInfoAccess_t *authInfo;
int32 rc;
int num_ocsp_uris = 0;
int total_uri_len = 0;
psCryptoOpen(PS_CRYPT_CONFIG);
rc = psX509ParseCertFile(NULL, "mycert.pem", &cert,
    CERT_STORE_UNPARSED_BUFFER);
if (rc == PS_SUCCESS) {
    if (cert->extensions.authorityInfoAccess != NULL) {
        authInfo = cert->extensions.authorityInfoAccess;
        do {
            if (authInfo->ocsp != NULL) {
                num_ocsp_uris++;
                total_uri_len += authInfo->ocspLen;
            }
            authInfo = authInfo->next;
        } while (authInfo);
    }
}
psX509FreeCert(cert);
```

3 CERTIFICATE REVOCATION LISTS

CRL is an X.509 format for publishing lists of certificate files that have been revoked. A certificate file that has been revoked should not be trusted for use in PKI environments.

3.1 Overview

Retrieving and maintaining a collection of CRLs is the responsibility of the application that is choosing to trust the remote certificate of a peer. In standard SSL connections, this application is the client because that is the peer that is placing trust in the certificate that is sent from the server. In SSL connections that use client authentication, the server could also maintain CRLs to authentication the certificates that are sent by the client.

3.2 X.509 Certificates and CRL Distribution Points

The location of where to retrieve a CRL is embedded within X.509 certificates in the `cRLDistributionPoints` extension. The Matrix CRL implementation only supports Internet web URL formats that have been encoded as a `uniformResourceIdentifier` within the `fullName` of the `distributionPoint`.

It may be helpful to note that distribution points encoded within an X.509 certificate are references to where the CRL is located that could potentially revoke that very certificate. In other words, the certificate is not specifying the CRL location of where to find CRLs that it itself issues. This should make sense when you consider that the content of an X.509 certificate is created by the issuing certificate. So the issuing CA certificate is placing the CRL distribution location into the subject cert of where that CA intends to store the CRL for the certificates it issues.

Matrix CRL makes the assumption that the issuer of an X.509 certificate is also the issuer of the CRL for a given subject certificate.

3.3 Parsing and Authentication of a CRL

A CRL is a signed data format. The parsing of a CRL will result in a list of certificates that have been revoked but trust in that list should be limited if the CRL itself has not been authenticated. The process of parsing and authenticating a CRL are two distinct operations.

A CRL may be parsed and stored in Matrix data formats without being authenticated. During revocation testing, the status of the CRL authentication will be reported independently from the revocation status of the subject certificate.

3.4 MatrixSSL and CRL

MatrixSSL provides an API to load and manage CRLs in a global cache that will be used when a certificate expects to have a revocation test. Fetching and loading the CRLs into the cache is currently a manual process that is discussed in the following sections.

MatrixSSL will attempt to perform a CRL authentication and certificate revocation test for any certificate that contains a `cRLDistributionPoints` extension while that certificate is being processed by the `psX509AuthenticateCert` function. There are several possible outcomes when performing those tests and the only status that will cause the authentication to fail and immediately return with an error is if the certificate has been revoked by an authenticated CRL. Other combinations of the test will set the `revokedStatus` member of the `psX509Cert_t` structure and the remainder of the authentication process will occur.

In all cases, the application SHOULD look to the `revokedStatus` members of the certificate chain when control is passed to the certificate callback function to determine the revocation status and apply application policy.

Due to CRL implementation details the SSL peer will often not have information about a CRL distribution point at the time of the initial SSL connection. As mentioned in the previous section, the CRL distribution location embedded in a certificate is where to find the CRL that could potentially revoke that very certificate. So when an end-entity certificate from a server is received for the first time, it is highly likely that the application is seeing the CRL distribution point for the first time so will not have a CRL loaded for that certificate. The CRL revocation status will be returned in the certificate callback for the application to process.

The dilemma of not holding a CRL during connection time poses an interesting implementation problem for CRL support in SSL connections. Do you go out and fetch the CRL via HTTP in the certificate callback while in the middle of the handshake? Or do you fail the handshake to go fetch the CRL and retry the SSL connection later? The `./apps/ssl/client.c` application has examples for both of these solutions and is discussed in the following sections.

3.5 Example application CRL support

The `./apps/ssl/client.c` application demonstrates how an SSL application can manage CRL. There are three points of integration that are described here. The following sections are API documentation for this integration.

1. At initialization time, the CA files are parsed to look for any CRL distribution points. If any are found, the CRL will be fetched via HTTP. Each CRL will then be parsed into the Matrix format and loaded into the global CRL cache. At this time, the list of CA files will also be used to attempt to authenticate any of the CRLs that were fetched. It is not a requirement that a CRL be authenticated prior to an SSL connection but it would be an optimization if that authentication was complete during initialization.
2. The next point of integration for CRL occurs within the certificate callback function (`certCb`) after the internal CRL test was made against the global cache. The primary focus of the certificate callback implementation is to confirm the authentication of the server certificate chain and the “alert” value of that function is processed first to see if there are any issues. If there are no authentication problems, the revocation status processing is hit near the bottom of the function where each certificate in the chain has its “`revokedStatus`” examined. These are the status options that would require an action.
 - a. `CRL_CHECK_EXPECTED` is an indication that the certificate has a CRL distribution point but there was no CRL found in the global cache to test against. Here is where we make the design decision to either fetch the CRL right there in the middle of the handshake or to fail the handshake to fetch the CRL and attempt the connection again. The compile time define set in `client.c` to determine the path is `MIDHANDSHAKE_CRL_FETCH`. In the midhandshake case, the set of server certs are passed to a function for processing and then the “`revokedStatus`” of each cert is examined again. There is a sanity check in the code so the CRL fetch will only be attempted one time. In the case where the handshake is failed to fetch the CRLs, the distribution point URLs are pulled out of the server certificates and saved aside. The current handshake is failed with the alert `CERTIFICATE_UNKNOWN`.
 - b. `CRL_CHECK_REVOKED_AND_AUTHENTICATED` is an indication that we truly have a revoked certificate. Return the `CERTIFICATE_REVOKED` alert to stop the handshake.
 - c. `CRL_CHECK_REVOKED_BUT_NOT_AUTHENTICATED` is an indication that we found the certificate was revoked in a CRL that is stored in the cache but that CRL has not yet been authenticated. Return the `CERTIFICATE_REVOKED` alert to stop the handshake.

- d. CRL_CHECK_CRL_EXPIRED is an indication that the cache held the correct CRL for the certificate but is no longer valid. Here, the example removes the CRL from the cache and moves to the same logic as CRL_CHECK_EXPECTED as though the CRL never existed.
3. In the CRL_CHECK_EXPECTED case where we have decided to fail the connection to retrieve the CRL at a later time, the next point of integration can be found where the function `fetchSavedCRL` is called. Note that the reconnection attempt will only occur if the example client has been configured to attempt multiple connections to begin with. This is controlled with the “-n” command line option that sets the number of connection attempts.

3.6 Example application API

Matrix packages do not currently provide APIs that support specific socket implementations. For example, the MatrixSSL API is “buffer based” and requires the application to send and receive data buffers using a socket implementation of their choosing. Supporting CRL requires that the application connect to an HTTP server and retrieve a CRL over sockets. This functionality is not provided within the supported APIs of the Matrix package. However, the `./apps/ssl/client.c` example application has implemented this functionality that will provide a useful template for other applications that wish to add CRL support.

3.6.1 `fetchCRL`

```
int32_t fetchCRL(psPool_t *pool, char *url, uint32_t urlLen,
unsigned char **crlBuf, uint3_t *crlBufLen);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where allocations will be made. NULL if not needed
url	input	The fully formed URL of the CRL to fetch (http://www.crlstore.com/mycrl.crl)
urlLen	input	The byte length of url
crlBuf	output	On success, the DER stream of the CRL that was fetched
crlBufLen	output	The byte length of crlBuf

Return Value	Description
PS_SUCCESS	

This function is an example implementation to fetch a given CRL using an HTTP GET request. The implementation uses blocking POSIX sockets to connect to a web server, send the GET request, parse the reply and store the CRL in a newly allocated buffer.

The caller must free the CRL buffer returned in `crlBuf` with `psFree`.

There are known limitations with the parsing of the host response. The parsing code is not a robust HTTP implementation and is looking for specific strings in the reply to determine success and CRL binary size. If a `recv()` call happens to fall in the middle of one of these strings the behaviour is undefined.

3.6.2 `fetchParseAndAuthCRLfromCert`

```
int32_t fetchParseAndAuthCRLfromCert(psPool_t *pool, psX509Cert_t *cert,
psX509Cert_t *potentialIssuers);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where allocations will be made. NULL if not needed
cert	input	The output CRL structure. Must be freed by caller
potentialIssuers	input	The DER formatted CRL stream

Return Value	Description
PS_SUCCESS	

Given a certificate, this function combines the functionality of `fetchCRL` with the supported Matrix CRL API to parse and load CRLs into the global cache and then attempt to authenticate the CRL.

The `cert` parameter may be a certificate chain. In that case, each certificate in the chain will be checked for a CRL distribution point and put through the process of fetch, parse, load to cache, and authentication attempt.

For the authentication phase, the `potentialIssuers` chain will be used for each retrieved CRL to attempt an authentication. The CA files of a client are often the set that would be expected to be issuers. In addition, any certificates in the subject certificate chain itself will also be tested as issuers. Checking the certificate chain itself is a useful feature to support servers that send certificate chains of 2 or more certificates where a parent is also likely the CRL issuer.

This function is used in two places in the `client.c` example.

1. At initialization time, the list of CA files is passed through to see if the global cache can be seeded with any known CRLs
2. In the midhandshake fetch implementation, the server certificates are passed to this function during the certificate callback.

3.6.3 `fetchParseAndAuthCRLfromUrl`

```
int32_t fetchParseAndAuthCRLfromUrl(psPool_t *pool, unsigned char *url,
                                     uint32_t urlLen, psX509Cert_t *potentialIssuers);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where allocations will be made. NULL if not needed
url	input	The fully formed URL of the CRL to fetch (http://www.crlstore.com/mycrl.crl)
urlLen	input	The byte length of url
potentialIssuers	input	The DER formatted CRL stream

Return Value	Description
PS_SUCCESS	

Given a URL, this function combines the functionality of `fetchCRL` with the supported Matrix CRL API to parse and load CRLs into the global cache and then attempt to authenticate the CRL.

For the authentication phase, the `potentialIssuers` chain will be used for each retrieved CRL to attempt an authentication. The CA files of a client are often the set that would be expected to be issuers.

This function is used for the example where the handshake is failed so that the application can go out and fetch the CRLs for later connection attempts. The URL in this case will have been saved aside during the certificate callback to be used here.

3.7 Supported CRL API

These supported functions are implemented as part of the supported crypto package of Matrix distributions and will enable applications to manage CRLs.

3.7.1 psX509Crl_t data type

```
typedef struct x509revoked {
    unsigned char    *serial;
    uint16_t         serialLen;
    struct x509revoked *next;
} x509revoked_t;

typedef struct psCRL {
    psPool_t         *pool;
    int32_t           authenticated; /* Has this CRL been authenticated */
    unsigned char     sigHash[MAX_HASH_SIZE];
    int32_t           sigHashLen;
    int32_t           nextUpdateType;
    char              *nextUpdate; /* Only concerned about expiration */
    int32_t           sigAlg;
    unsigned char     *sig;
    uint16_t          sigLen;
    uint16_t          expired;
    x509DNattributes_t issuer;
    x509v3extensions_t extensions;
    x509revoked_t     *revoked;
    struct psCRL       *next;
} psX509Crl_t;
```

3.7.2 psX509ParseCRL

```
int32_t psX509ParseCRL(psPool_t *pool, psX509Crl_t **crl,
    unsigned char *crlBin, int32_t crlBinLen);
```

Parameter	Input/Output	Description
pool	input	Optional memory pool to where the optput CRL will be allocated
crl	output	The output CRL structure. Must be freed by caller
crlBin	input	The DER formatted CRL stream
crlBinLen	input	Byte length of crlBin

Return Value	Description
PS_SUCCESS	Success. A valid CRL structure is allocated and populated in "crl" parameter
PS_MEM_FAIL	Failure. Unable to allocate memory for the structure
PS_PARSE_FAIL	Failure. Unable to parse CRL stream
PS_ARG_FAIL	Failure. Bad input parameters

Parses a CRL DER stream into a Matrix `psX509Crl_t` structure.

3.7.3 psCRL_Update

```
int psCRL_Update(psX509Crl_t *CRL, int deleteExisting);
```

Parameter	Input/Output	Description
CRL	input	A parsed CRL to be added to the global CRL cache
deleteExisting	input	1 to replace an existing CRL if found. 0 to append

Return Value	Description
0	Failure. The CRL parameter was NULL or already existed in cache
1	Success. CRL was added to cache

Add the CRL reference to the global CRL cache.

IMPORTANT: A direct reference to the supplied pointer is stored in the cache. A copy of the CRL is not added to the global cache. Therefore, if the CRL is deleted in the future via `psCRL_Delete` or `psX509FreeCRL` it will be removed from the cache and the memory freed.

3.7.4 psCRL_determineRevokedStatus

```
int32_t psCRL_determineRevokedStatus(psX509Cert_t *cert);
```

Parameter	Input/Output	Description
cert	input	The cert to test for the revoked status

Return Value	Description
0	Failure. NULL cert parameter
CRL_CHECK_CRL_EXPIRED	Success. CRL is found but has expired. No revoked tests were run on certificate
CRL_CHECK_PASSED_AND_AUTHENTICATED	Success. Authenticated CRL was found for this certificate and the certificate has not been revoked
CRL_CHECK_PASSED_BUT_NOT_AUTHENTICATED	Success. CRL was found for this certificate and the certificate has not been revoked BUT the CRL has not been authenticated
CRL_CHECK_REVOKED_AND_AUTHENTICATED	Success. Authenticated CRL was found for this certificate and the certificate has been revoked
CRL_CHECK_REVOKED_BUT_NOT_AUTHENTICATED	Success. CRL was found for this certificate and the certificate has been revoked BUT the CRL has not been authenticated
CRL_CHECK_EXPECTED	Success. No CRL was found in the global CRL cache but the certificate had a CRL distribution point. The CRL should be fetched
CRL_CHECK_NOT_EXPECTED	Success. No CRL was found in the global CRL cache but this certificate did not have a CRL distribution point so a CRL probably does not exist for this certificate.

Run the given certificate through the revocation tests. The value of `revokedStatus` of the `psX509Cert_t` structure will be set to whatever the return value of this function call is.

3.7.5 psCRL_Delete

```
int psCRL_Delete(psX509Crl_t *CRL);
```

Parameter	Input/Output	Description
CRL	input	The CRL to be deleted from the global CRL cache

Return Value	Description
0	Failure. The CRL parameter was NULL or didn't exist
1	Success. CRL was deleted from the cache

Delete a CRL from the global CRL cache and free the memory of the CRL. This function has the same behaviour as `psX509FreeCRL` but was given a `psCRL_` prefix to create a consistent set of functions that manage the global CRL cache.

3.7.6 psCRL_DeleteAll

```
void psCRL_DeleteAll();
```

Deletes all CRLs from the global CRL cache and frees the memory for each CRL.

3.7.7 psCRL_Remove

```
int psCRL_Remove(psX509Crl_t *CRL);
```

Parameter	Input/Output	Description
CRL	input	The CRL to be removed from the global CRL cache

Return Value	Description
0	Failure. The CRL parameter was NULL or didn't exist
1	Success. CRL was removed from the cache

Remove a CRL from the global CRL cache but do not free the memory of the CRL. The CRL may be deleted with `psX509FreeCRL` at a later time.

3.7.8 psCRL_RemoveAll

```
void psCRL_RemoveAll();
```

Removes all CRLs from the global CRL cache but does not free the memory. Each managed CRLs may be deleted with `psX509FreeCRL` at a later time.

3.7.9 psX509FreeCRL

```
void psX509FreeCRL(psX509Crl_t *crl);
```

Parameter	Input/Output	Description
crl	input	The CRL structure to free

Free a CRL structure allocated by psX509ParseCRL. If the CRL entry had been added to the global CRL cache with psCRL_Update or psCRL_Insert it will be deleted from the cache.

3.7.10 psCRL_GetCRLForCert

```
psX509Crl_t* psCRL_GetCRLForCert(psX509Cert_t *cert);
```

Parameter	Input/Output	Description
cert	input	The cert used to search for the associated CRL

Return Value	Description
NULL	Failure. No matching CRL was found
<valid pointer>	Success. CRL is found and referenced by returned pointer

Locate a CRL for a given certificate in the global CRL cache. This function is useful to locate a CRL from the global cache to be deleted when the certificate is reporting a value of CRL_CHECK_CRL_EXPIRED as the revokedStatus status.

3.7.11 psX509GetCRLdistURL

```
int32_t psX509GetCRLdistURL(psX509Cert_t *cert, char **url, uint32_t *urlLen);
```

Parameter	Input/Output	Description
cert	input	A parsed certificate from which to search for the CRL distribution point
url	output	A pointer to the URL distribution point or NULL if not found
urlLen	output	Byte length of url

Return Value	Description
PS_TRUE	Success. The url parameter will point to the URL distribution point of the CRL
PS_FALSE	Failure. Certificate did not contain a URL distribution point for a CRL
PS_ARG_FAIL	Failure. Bad input parameters

Return the URL of where to find the CRL for a given certificate. The url value will point directly into the read-only psX509Cert_t structure and should not be destructively parsed or freed.

A known limitation is that this function will only return the first distribution point that is found in a certificate.

3.7.12 psX509AuthenticateCRL

```
int32_t psX509AuthenticateCRL(psX509Cert_t *CA, psX509Crl_t *CRL,
void *userPtr);
```

Parameter	Input/Output	Description
CA	input	The issuing certificate of the CRL that will be used to authenticate the CRL signature
CRL	input	The CRL to authenticate
userPtr	input	NULL or user context for internal memory pool usage

Return Value	Description
PS_SUCESS	Success. CRL is authenticated and the "authenticated" member has been set to PS_TRUE
PS_CERT_AUTH_FAIL_EXTENSION	Failure. Certificate extensions did not match what the CRL reported as the issuer
PS_CERT_AUTH_FAIL_DN	Failure. Certificate name did not match what the CRL reported as the issuer
PS_UNSUPPORTED_FAIL	Failure. Signature algorithm of CRL is not supported
PS_CERT_AUTH_FAIL_SIG	Failure. Signature authentication failed
PS_MEM_FAIL	Failure. Memory allocation error
PS_ARG_FAIL	Failure. Bad input parameters

Performs the authentication tests on a CRL given an issuer. If the authentication is successful the `authenticated` member of the CRL structure will be set to `PS_TRUE` (1).

This function will always attempt to perform the authentication so the `authenticated` member of the CRL will be reset to 0 at the beginning of this function regardless of the current value.

This function is internally invoked as part of the `psX509AuthenticateCert` logic to handle use cases where a server has presented a certificate chain and the parent certificate will only be available internally during that handshake time window.