MatrixSSL Developer's Guide

MatrixSSL 3.1

PeerSec Networks, Inc. 410 Broadway Ave E. #205 Seattle, WA 98102 **T** 425.646.7850 **F** 206.501.4366

support@peersec.com www.peersec.com

Overview

This developer's guide is a general SSL/TLS overview and a MatrixSSL specific integration reference for adding SSL security into an application.

Who Is This Document For?

- Software developers that are securing applications with MatrixSSL
- Anyone wanting to learn more about MatrixSSL
- Anyone wanting to learn more about the SSL/TLS protocol

Documentation Style Conventions

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

Commercial Version Differences

Some of the compile options, functions, and structures in this document provide additional features only available in the commercially licensed version of MatrixSSL. Sections of this document that refer to the commercial version will be shaded.

Security Considerations

Prior to working directly with the MatrixSSL library there are a couple SSL security concepts that application integrators should be familiar with.

SSL vs. TLS

MatrixSSL supports both the TLS and SSL protocols. Despite the difference in acronym, TLS 1.0 is simply version 3.1 of SSL. There are no practical security differences between the protocols, and only minor differences in how they are implemented. It was felt that 'Transport Layer Security' was a more appropriate name then 'Secure Sockets Layer' going forward beyond SSL 3.0. In this documentation, the term SSL is used generically to mean SSL/TLS, and TLS is used to indicate specifically the TLS protocol. Looking to the future, the specifications for both TLS 1.1 and 1.2 are available, but not widely deployed today.

Selecting Cipher Suites

The strength (and thus performance) of the secure communications are primarily determined by the choice of cipher suites that will be supported. A cipher suite determines how two peers progress through an SSL handshake as well as how the final application data will be encrypted over the secure connection. The four components of any given cipher suite are **key exchange**, **authentication, encryption** and **digest hash**.

Key exchange mechanisms refer to how the peers agree upon a common symmetric key that will be used to encrypt data after handshaking is complete. The two common key exchange algorithms are RSA and Diffie-Hellman (DH). Currently, when Diffie-Hellman is chosen it is used almost exclusively in ephemeral mode (DHE) in which new private key pairs are generated for each connection to allow perfect forward secrecy. The tradeoff for DHE is a much slower SSL handshake as key generation is a relatively processor-intensive operation. Some older protocols also specify DH, as it was the first widely publicized key exchange algorithm.

The authentication algorithm specifies how the peers will prove their identities to each other. Authentication options within cipher suites are RSA, DSA, Elliptic Curve DSA (ECDSA), Pre-shared Key (PSK), or anonymous if no authentication is required. RSA has the unique property that it can be used for both key exchange and authentication. For this reason, RSA has become the most widely implemented cipher suite mechanism for SSL communications. RSA keys strengths of between 1024 and 2048 bits are the most common.

The encryption component of the cipher suite identifies which symmetric cipher is to be used when exchanging data at the completion of the handshake. The AES block cipher is recommended for new implementations, and is the most likely to have hardware acceleration support.

Finally, the digest hash is the choice of checksum algorithm used to confirm the integrity of exchanged data, with SHA-1 being the most common. Here is a selection of cipher suites that illustrate how to identify the four components.

Cipher Suite	Key Exchange	Auth Type	Encryption	Digest Hash
`SSL_RSA_WITH_3DES_EDE_CBC_SHA`	RSA	RSA	3DES	SHA-1
`SSL_DH_anon_WITH_RC4_128_MD5`	DH	anonymous	RC4-128	MD5
`TLS_RSA_WITH_AES_128_CBC_SHA`	RSA	RSA	AES-128	SHA-1
`TLS_DHE_RSA_WITH_AES_256_CBC_SHA`	DHE	RSA	AES-256	SHA-1
`TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA`	ECDHE	RSA	AES-128	SHA-1
`TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA`	ECDHE	ECDSA	AES-256	SHA-1

Authentication Mode

By default in SSL, it is the server that is authenticated by a client. It is easiest to remember this when thinking about purchasing a product online with a credit card over an HTTPS (SSL) connection. The client Web browser must authenticate the server in order to be confident the credit card information is being sent to a trusted source. This is referred to as one-way authentication or server authentication and is performed as part of all standard SSL connections (unless, of course, a cipher suite with an authentication type of anonymous has been agreed upon).

However, in some use-case scenarios the user may require that both peers authenticate each other. This is referred to as mutual authentication or client authentication. If the project requires client authentication there is an additional set of key material that must be used to support it.

Certificates and Private Keys

With a cipher suite and authentication mode chosen, the user will need to obtain or generate the necessary key material for supporting the authentication and key exchange mechanisms. X.509 is the standard for how key material is stored in **certificate files**.

The peer that is being authenticated must have a private key and a public certificate. The peer performing the authentication must have the Certificate Authority (CA) certificate that was used to issue the public certificate. In the standard one-way authentication scenario this means the server will load a private key and certificate while the client will load the CA file.

If client authentication is needed the mirror image of CA, certificate, and private key files must also be used. This chart shows which files are needed by clients and servers that are using a standard RSA based cipher suite such as `SSL_RSA_WITH_3DES_EDE_CBC_SHA`.

Authentication Mode	Server Key Files	Client Key Files
One-way server authentication	1. RSA server certificate file 2. RSA private key file for the server certificate file	1. Certificate Authority certificate file that issued the server certificate
Additional for client authentication	3. Certificate Authority certificate file that issued the client certificate	2. RSA client certificate file 3. RSA private key file for the client certificate file

For information on how to create Certificate Authority root and child certificates please see the PeerSec Key Generation Utilities document.

Application Integration Flow

MatrixSSL is a C code library that provides a security layer for client and server applications allowing them to securely communicate with other SSL enabled peers. MatrixSSL is transport agnostic and can just as easily integrate with an HTTP server as it could with a device communicating through a serial port. For simplicity, this developer's guide will assume a socket based implementation for all its examples unless otherwise noted.

The term *application* in this document refers to the peer (client or server) application the MatrixSSL library is being integrated into.

This section will detail the specific points in the application life-cycle where MatrixSSL should be integrated. In general, MatrixSSL APIs are used for initialization/cleanup, when new secure connections are being established (handshaking), and when encrypting/decrypting messages exchanged with peers.

Refer to the MatrixSSL API document to get familiar with the interface to the library and with the example code to see how they are used at implementation. Follow the guidelines below when using these APIs to integrate MatrixSSL into an application.

ssl_t Structure

The `ssl_t` structure holds the state and keys for each client or server connection as well as buffers for encoding and decoding SSL data. The buffers are dynamically managed internally to make the integration with existing non-secure software easier. SSL is a record based protocol, and the internal buffer management makes a better "impedance match" with classic stream based protocols. For example, data may be read from a socket, but if a full SSL record has not been received, no data is available for the caller to process. This partial record is held within the `ssl_t` buffer. The MatrixSSL API is also designed so there are no buffer copies, and the caller is able to read and write network data directly into the SSL buffers, providing a very low memory overhead per session.

Initialization

MatrixSSL must be initialized as part of the application initialization with a call to `matrixSslOpen`. This function takes no parameters and sets up the internal structures needed by the library.

In most cases, the application will subsequently load the key material from the file system. RSA certificates, Diffie-Hellman parameters, and Pre-Shared Keys for the specific peer application must be parsed before creating a new SSL session. The `matrixSslNewKeys` function is used to allocate the key storage and `matrixSslLoadRsaKeys`, `matrixSslLoadDhParams`, and `matrixSslLoadPsk` are used to parse the key material into the `sslKeys_t` structure during initialization. The populated key structure will be used as an input parameter to `matrixSslNewClientSession` or `matrixSslNewServerSession`.

The allocation and loading of the `sslKeys_t` structure is most commonly done a single time at start and the application uses those keys for each connection. Alternatively, a new `sslKeys_t` structure can be allocated once for each secure connection and freed immediately after the connection is closed. This should be done if the application has multiple certificate files depending on the identity of the connecting entity or if there is a security concern with keeping the RSA keys in memory for extended periods of time.

Once the application is done with the keys, the associated memory is freed with a call to `matrixSslFreeKeys`.

Creating a Session

The next MatrixSSL integration point in the application is when a new session is starting. In the case of a client, this is whenever it chooses to begin one, because SSL is a client initiated protocol (like HTTP). In the case of a server, a new session should be started when the server accepts an incoming connection from a client. In a socket based application, this would typically happen when the `accept` socket call returns with a valid incoming socket. The application sets up a new session with the API `matrixSslNewClientSession` or `matrixSslNewServerSession`. The returned `ssl_t` context will become the input parameter for the public APIs that act at a session level.

The required input parameters to the session creation APIs differ based on whether the application is assuming a server or client role. Both require a populated keys structure (discussed in the previous section) but a client can also nominate a specific cipher suite or session ID when starting a session. The ciphers that the server will accept are determined at compile time.

The client should also always nominate a certificate callback function during `matrixSslNewClientSession`. This callback function will be invoked mid-handshake to allow the user to inspect the key material, date and other certificate information sent from the server. For detailed information on this callback function, see the API documentation for `matixSslNewClientSession`.

In the commercial version the server may also choose to nominate a certificate callback function if client authentication is desired. The MatrixSSL library must have be compiled with `USE_CLIENT_AUTH` defined in order to use this parameter.

For clients wishing to quickly (and securely) reconnect to a server that it has recently connected to, there is an optional `sessonId` parameter that may be used to initiate a faster resumed handshake (the cpu intensive public key exchange is omitted). To use the session parameter, a client should allocate a `sslSessionId_t` structure, initialize it with `matrixSslInitSessionId` and pass its pointer to `matrixSslNewClientSession` during the initial connection with the server. Over the course of the session negotiation, the MatrixSSL library will populate that structure behind-the-scenes so that during the next connection the same `sessionId` parameter address can be used to initiate the resumed session.

Handshaking

During client session initialization with `matrixSslNewClientSession` the SSL handshake message CLIENT_HELLO is encoded to the internal outgoing buffer. The client now needs to send this message to the server over a communication channel.

The sequence of events that should always be used to transmit pending handshake data is as follows:

1. The user calls `matrixSslGetOutdata` to retrieve the encoded data and number of bytes to be sent
2. The user sends the # of bytes indicated from the out data buffer pointer to the peer
3. The user calls `matrixSslSentData` with the actual number of bytes that were sent
4. If more data remains (bytes sent < bytes to be sent), repeat the above 3 steps when transport layer is ready to send again

When the server receives notice that a client is starting a new session the `matrixSslNewServerSession` API is invoked and the incoming data is retrieved and processed.

The sequence of events that should always be used when expecting handshake data from a peer is as follows:

1. The application calls `matrixSslGetReadbuf` to retrieve a pointer to available buffer space in the `ssl_t` structure.
2. The application reads incoming data into that buffer
3. The application calls `matrixSslReceivedData` to process the data
4. The application examines the return code from `matrixSslReceivedData` to determine the next step

All incoming messages should be copied into the provided buffer and passed to `matrixSslReceivedData` which processes the message and drives the handshake through the built-in SSLv3 or TLS state machine. The parameters include the SSL context and the number of bytes that have been received. Its return code tells the application what the message was and how it is to be handled.

`MATRIXSSL_REQUEST_SEND`	Success. The processing of the received data resulted in internal creation of an SSL response message that needs to be sent to the peer. If this return code is hit the user should call `matrixSslGetOutdata` to retrieve the encoded outgoing data and send it.
`MATRIXSSL_REQUEST_RECV`	Success. More data must be received and this function must be called again. Most likely a partial record was read and more data is required to continue parsing. User must call `matrixSslGetReadBuf` again to receive the updated buffer pointer and length to where the remaining data should be read into.

`MATRIXSSL_HANDSHAKE_COMPLETE`	Success. The SSL handshake is complete. This return code is returned to client side implementation during a full handshake after parsing the FINISHED message from the server. It is possible for a server to receive this value if a resumed handshake is being performed where the client sends the final FINISHED message. Client applications will typically use this state as a trigger to send a client initiated protocol message (such as HTTP GET).
`MATRIXSSL_RECEIVED_ALERT`	Success. The data that was processed was an SSL alert message. In this case, the `plainText` pointer will be two bytes (`ptLen` will be 2) in which the first byte will be the alert level and the second byte will be the alert description. After examining the alert, the user must call `matrixSslProcessedData` to indicate the alert was processed and the data may be internally discarded.
`MATRIXSSL_APP_DATA`	Success. The data that was processed was application data that the user should process. In this return code case the `plainText` and `ptLen` output parameters will be valid. The user must process the all the data directly from plainText or copy it aside for later processing. After handling the data the user must call `matrixSslProcessedData` to indicate the plainText data may be internally discarded.
`< 0`	Failure. See API documentation for more information

Communicating Securely With Peers

Once the handshake is complete the application wishing to encrypt data to be sent to the peer should follow these steps:

1. The application first determines the length of the plaintext that needs to be sent
2. The application calls `matrixSslGetWritebuf` with that length to retrieve a pointer to an internally allocated buffer.
3. The application writes the plaintext into the buffer and then calls `matrixSslEncodeWritebuf` to encrypt the plaintext
4. The application calls `matrixSslGetOutdata` to retrieve the encoded data and length to be sent (SSL always adds some overhead to the message size)

5. The application sends the out data buffer contents to the peer.
6. The application calls `matrixSslSentData` with the # of bytes that were actually sent

The sequence of events that should always be used when expecting application data from a peer is as follows:

1. The application calls `matrixSslGetReadbuf` to retrieve an allocated buffer
2. The application copies the incoming data into that buffer
3. The application calls `matrixSslReceivedData` to process the data
4. The application confirms the return code from `matrixSslReceivedData` is `MATRIXSSL_APP_DATA` and parses `ptLen` bytes of the returned `plainText`
5. If the return code does not indicate application data, handle the return code as described in the handshaking section above.
6. The application calls `matrixSslProcessedData` to inform the library it is finished with the plaintext and checks to see if there are additional records in the buffer to process.

Ending a Session

When the application receives notice that the session is complete or has determined itself that the session is complete, it should notify the other side, close the socket and delete the session. This is done by calling `matrixSslEncodeClosureAlert` and `matrixSslDeleteSession`.

A call to `matrixSslEncodeClosureAlert` is an optional step that will encode an alert message to pass along to the other side to inform them to close the session cleanly. The closure alert buffer is retrieved and sent using the same `matrixSslGetOutdata`/`matrixSslSentData` mechanism that all outgoing data uses. Since the connection is being closed, the application shouldn't block indefinitely on sending the closure alert.

Closing the Library

At application exit the MatrixSSL library should be un-initialized with a call to `matrixSslClose`. If the application has called `matrixSsNewKeys` as part of the initialization process and kept its keys in memory it should call `matrixSslDeleteKeys` before calling `matrixSslClose`. Also, any existing SSL sessions should be freed by calling `matrixSslDeleteSession` before calling `matrixSslClose`.

Working implementations of MatrixSSL client and server applications integration can be found in the *apps* subdirectory of the distribution package.

Configurable Features

Functionality Defines

MatrixSSL contains a set of optional features that are configurable at compile time. This allows the user to remove unneeded functionality to reduce their application's footprint. Each of these options are pre-processor defines that can be disabled by simply commenting out the `#define` in the specified header files or by using the -D flag in the build environment. APIs with dependencies on optional features contain a **Define Dependencies** section in the documentation for that function.

`PS_USE_FILE_SYSTEM`	Define in build environment	Enables file access for parsing X.509 certificates and private keys.
`ENABLE_SECURE_REHANDSHAKES`	*matrixsslConfig.h*	Enable secure re-handshaking as defined in RFC 5746
`REQUIRE_SECURE_REHANDSHAKES`	*matrixsslConfig.h*	Halt communications with any SSL peer that has not implemented RFC 5746
`ENABLE_INSECURE_REHANDSHAKES`	*matrixsslConfig.h*	Enable legacy renegotiations. NOT RECOMMENDED
`USE_MULTITHREADING`	*coreConfig.h*	Enables mutex support in the core module for internal locking of shared resources.
`HAVE_NATIVE_INT64`	*coreConfig.h*	Enable if the platform has a native 64-bit data type (long long).

`USE_PEERSEC_MEMORY_MANAGEMENT`	*coreConfig.h*	Enables the deterministic memory management module. See the specific documentation for this feature.
`USE_CLIENT_SIDE_SSL`	*matrixsslConfig.h*	Enables client side SSL support
`USE_SERVER_SIDE_SSL`	*matrixsslConfig.h*	Enables server side SSL support
`USE_CLIENT_AUTH`	*matrixsslConfig.h*	Enables two-way(mutual) authentication
`USE_TLS`	*matrixsslConfig.h*	Enables TLS protocol support
`USE_PRIVATE_KEY_PARSING`	*cryptoConfig.h*	Enables X.509 private key parsing
`USE_PKCS5`	*cryptoConfig.h*	Enables the parsing of encrypted X.509 private keys
`USE_CERT_PARSE`	*cryptoConfig.h*	Servers may optionally disable (if not using client auth) to exclude X.509 certificate parsing and reduce the application binary size.

Debug Configuration

MatrixSSL contains a set of optional debug features that are configurable at compile time. Each of these options are pre-processor defines that can be disabled by simply commenting out the `#define` in the specified header files.

`HALT_ON_PS_ERROR`	*coreConfig.h*	Enables the `osdepBreak` platform function whenever a `_psError` trace function is called. Helpful in debug environments.
`USE_CORE_TRACE`	*coreConfig.h*	Enables the `psTraceCore` family of APIs that display function-level messages in the core module

`USE_CRYPTO_TRACE`	*cryptoConfig.h*	Enables the `psTraceCrypto` family of APIs that display function-level messages in the crypto module
`USE_SSL_HANDSHAKE_MSG_TRACE`	*matrixsslConfig.h*	Enables SSL handshake level debug trace for troubleshooting connection problems
`USE_SSL_INFORMATIONAL_TRACE`	*matrixsslConfig.h*	Enables SSL function level debug trace for troubleshooting connection problems

SSL Handshaking

Handshake Variations

The core of SSL security is the handshake protocol that allows two peers to authenticate and negotiate symmetric encryption keys. A handshake is defined by the specific sequence of SSL messages that are exchanged between the client and server. A collection of messages being sent from one peer to another is called a **flight**.

Standard Handshake

The standard handshake is the most common and allows a client to authenticate a server. There are four flights in the standard handshake.

```
CLIENT                                                       SERVER
        ---------------------- CLIENT_HELLO -------------------->

        <-------------------- SERVER_HELLO --------------------
        <--------------------- CERTIFICATE -----------------------
        <--------------- SERVER_HELLO_DONE ---------------

        --------------- CLIENT_KEY_EXCHANGE ------------->
        ---------------- CHANGE_CIPHER_SPEC --------------->
        -------------------------- FINISHED ------------------------->

        <--------------- CHANGE_CIPHER_SPEC ---------------
        <------------------------ FINISHED --------------------------
```

Clients

The client is the first to send and the last to receive. Therefore, a MatrixSSL implementation of a client must be testing for the `MATRIXSSL_HANDSHAKE_COMPLETE` return code from `matrixSslReceivedData` to determine when application data is ready to be encrypted and sent to the server.

When a client wishes to begin a standard handshake, `matrixSslNewClientSession` will be called with an empty `sessionId`, initialized with `matrixSslInitSessionId`.

Client Authentication

The client authentication handshake is only available in the commercial version.

The client authentication handshake allows a two-way authentication. There are four flights in the client authentication handshake.

```
CLIENT                                                          SERVER
          ---------------------- CLIENT_HELLO -------------------->

          <-------------------- SERVER_HELLO --------------------
          <--------------------- CERTIFICATE ----------------------
          <--------------- CERTIFICATE_REQUEST --------------
          <--------------- SERVER_HELLO_DONE ---------------

          ---------------------- CERTIFICATE ----------------------->
          --------------- CLIENT_KEY_EXCHANGE ------------->
          ------------------ CERTIFICATE_VERIFY ---------------->
          ---------------- CHANGE_CIPHER_SPEC -------------->
          -------------------------- FINISHED ------------------------>

          <--------------- CHANGE_CIPHER_SPEC ---------------
          <------------------------ FINISHED -------------------------
```

Clients

The client is the first to send and the last to receive. Therefore, a MatrixSSL implementation of a client must be testing for the `MATRIXSSL_HANDSHAKE_COMPLETE` return code from `matrixSslReceivedData` to determine when application data is ready to be encrypted and sent to the server.

In order to participate in a client authentication handshake, the client must have loaded a Certificate Authority file during the call to `matrixSslLoadRsaKeys`.

Servers

To prepare for a client authentication handshake the server must nominate a certificate and private key during the call to `matrixSslLoadRsaKeys`. The actual determination of whether or not to perform a client authentication handshake is made when nominating a certificate callback parameter when invoking `matrixSslNewServerSession`. If the callback is provided, a client authentication handshake will be requested.

Session Resumption

Session resumption enables a previously connected client to quickly resume a session with a server. Session resumption is much faster than other handshake types because public key authentication is not performed (authentication is implicit since both sides will be using secret information from the previous connection). This handshake types has three flights.

```
CLIENT                                                      SERVER
        ---------------------- CLIENT_HELLO -------------------->

        <-------------------- SERVER_HELLO ---------------------
        <--------------- CHANGE_CIPHER_SPEC ---------------
        <------------------------ FINISHED --------------------------

        ---------------- CHANGE_CIPHER_SPEC --------------->
        -------------------------- FINISHED ------------------------->
```

Clients

The client is the first and the last to send data. Therefore, a MatrixSSL implementation of a client must be testing for the `MATRIXSSL_HANDSHAKE_COMPLETE` return code from `matrixSslSentData` to determine when application data is ready to be encrypted and sent to the server.

The client initiates a session resumption handshake by reusing the same `sessionId_t` structure from a previously connected session when calling `matrixSslNewClientSession`.

Servers

The MatrixSSL server will cache a `SSL_SESSION_TABLE_SIZE` number of session IDs for resumption. The length of time a session ID will remain in the case is determined by `SSL_SESSION_ENTRY_LIFE`.

Re-Handshakes

A re-handshake is a handshake over a currently connected SSL session. A re-handshake may take the form of a standard handshake, a client authentication handshake, or a resumed handshake. Either the client or server may initiate a re-handshake.

The `matrixSslEncodeRehandshake` API is used to initiate a re-handshake. The three most common reasons for initiating re-handshakes are:

1. **Re-key the symmetric cryptographic material**
   Re-keying the symmetric keys adds an extra level of security for applications that require the connection be open for long periods of time or transferring large amounts of data. Periodic changes to the keys can discourage hackers who are mounting timing attacks on a connection.

2. **Perform a client authentication handshake**
   A scenario may arise in which the server requires that the data being exchanged is only allowed for a client whose certificate has been authenticated, but the original negotiation took place without client authentication. In order to do a client authenticated re-handshake the server must call *matrixSslEncodeRehandshake* with a certificate callback parameter.

3. **Change cipher spec**
   The cipher suite may be changed on a connected session using a re-handshake if needed. The client must call *matrixSslEncodeRehandshake* with the new `cipherSpec`.