# MatrixSSL Developer's Guide

## Version 3.7

---

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 - 2013 - All rights reserved

# TABLE OF CONTENTS

**1 OVERVIEW ........ 4**

1.1 Nomenclature ........ 4

**2 SECURITY CONSIDERATIONS ........ 5**

2.1 SSL/TLS Version Security ........ 5

2.2 Selecting Cipher Suites ........ 5

2.3 Authentication Mode ........ 8

2.4 Authentication and Key Exchange ........ 8

2.4.1 Server and Client Authentication ........ 8

2.4.2 Certificate Validation and Authentication ........ 8

**3 APPLICATION INTEGRATION FLOW ........ 11**

3.1 ssl_t Structure ........ 11

3.2 Initialization ........ 11

3.3 Creating a Session ........ 11

3.4 Handshaking ........ 12

3.5 Communicating Securely With Peers ........ 13

3.5.1 Encrypting Data ........ 13

3.5.2 Decrypting Data ........ 13

3.6 Ending a Session ........ 14

3.7 Closing the Library ........ 14

**4 CONFIGURABLE FEATURES ........ 15**

4.1 Protocol and Performance ........ 15

4.2 Public Key Math Assembly Optimizations ........ 16

4.3 Debug Configuration ........ 17

4.4 Minimum Firmware Configuration ........ 17

**5 SSL HANDSHAKING ........ 18**

5.1 Standard Handshake ........ 18

5.2 Client Authentication Handshake ........ 19

5.3 Session Resumption Handshake ........ 20

5.4 Other Handshakes ........ 20

5.5 Re-Handshakes ........ 20

5.5.1 Disable Re-Handshaking At Runtime ........ 21

5.5.2 The Re-Handshake Credit Mechanism ........ 21

**6 OPTIONAL FEATURES ........ 22**

6.1 Stateless Session Ticket Resumption ........ 22

6.2 Server Name Indication Extension ........ 22

6.3 Maximum Fragment Length ........ 23

6.4 Truncated HMAC ........ 23

6.5 Application Layer Protocol Negotiation Extension ........ 23

6.6 MatrixSSL Statistics Framework ........ 25

6.7 ZLIB Compression ....................................................................................25

# 1 OVERVIEW

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

This document is primarily intended for the software developer performing MatrixSSL integration into their custom application but is also a useful reference for anybody wishing to learn more about MatrixSSL or the SSL/TLS protocol in general.

For additional information on the APIs discussed here please see the MatrixSSL API document included in this package.

## 1.1 Nomenclature

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 than '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.  SSL 2.0 is deprecated and not supported. MatrixSSL supports SSL 3.0, TLS 1.0, TLS 1.1 and TLS 1.2 protocols. In addition, the DTLS protocol is based closely on TLS 1.1 and beyond.

# 2 SECURITY CONSIDERATIONS

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

## 2.1 SSL/TLS Version Security

Although TLS 1.0 and above can be considered secure, several weaknesses have been discovered in some versions and cipher combinations.

All of the issues discovered are mitigated by default in MatrixSSL. Additionally, SSL 3.0 is disabled by default in MatrixSSL to reduce version downgrade attacks and the padding oracle attack POODLE.

| Vulnerability | SSL 3.0 | TLS 1.0 | TLS 1.1 | TLS 1.2 | Fixed | Solution |
|---|---|---|---|---|---|---|
| BEAST | **Vuln** | **Vuln** | **Ok** | **Ok** | **Yes** | USE_BEAST_WORKAROUND enabled by default<br>http://en.wikipedia.org/wiki/Transport_Layer_Security#BEAST_attack<br>Some implementations of TLS are not compatible with this workaround. |
| CRIME | **Vuln** | **Vuln** | **Vuln** | **Vuln** | **Yes** | USE_ZLIB_COMPRESSION disabled by default<br>http://en.wikipedia.org/wiki/CRIME_(security_exploit) |
| BREACH | **Vuln** | **Vuln** | **Vuln** | **Vuln** | **Yes** | Application code should not compress frequently used headers<br>http://en.wikipedia.org/wiki/BREACH_(security_exploit) |
| LUCKY13 Padding Attacks | **Vuln** | **Vuln** | **Vuln** | **Vuln** | **Yes** | Internal blinding for block cipher padding automatically applied<br>http://en.wikipedia.org/wiki/Lucky_Thirteen_attack<br>Applies to block ciphers only. |
| POODLE | **Vuln** | **Ok** | **Ok** | **Ok** | **Yes** | SSL 3.0 disabled by default with DISABLE_SSL3 since version 3.3.1<br>http://en.wikipedia.org/wiki/POODLE |
| Renegotiation Attacks | **Vuln** | **Vuln** | **Vuln** | **Vuln** | **Yes** | REQUIRE_SECURE_REHANDSHAKES enabled by default<br>http://en.wikipedia.org/wiki/Transport_Layer_Security#Renegotiation_attack<br>Some implementations of TLS are not compatible with this extension |
| False Start Weakness | **Vuln** | **Vuln** | **Vuln** | **Vuln** | **Yes** | ENABLE_FALSE_START disabled by default<br>http://en.wikipedia.org/wiki/Transport_Layer_Security#Version_rollback_attacks |
| RC4 Weakness | **Vuln** | **Less Vuln** | **Less Vuln** | **Less Vuln** | **Yes** | USE_SSL_RSA_WITH_RC4_128_* disabled by default<br>Internal code limitations for the number of bytes RC4 will encode<br>http://en.wikipedia.org/wiki/Transport_Layer_Security#RC4_attacks<br>SSL 3.0 is more vulnerable because RC4 is more commonly used |
| 3DES Weakness | **Vuln** | **Less Vuln** | **Less Vuln** | **Less Vuln** | **Yes** | USE_SSL_RSA_WITH_3DES_EDE_CBC_SHA disabled by default<br>http://en.wikipedia.org/wiki/Transport_Layer_Security#RC4_attacks<br>SSL 3.0 is more vulnerable because 3DES is more commonly used |
| MD5 MAC Weakness | **Vuln** | **Less Vuln** | **Less Vuln** | **Less Vuln** | **Yes** | USE_SSL_RSA_WITH_RC4_128_MD5 disabled by default<br>All other MD5 based ciphers disabled by default |
| MD5 Cert Weakness | **Vuln** | **Less Vuln** | **Less Vuln** | **Less Vuln** | **Yes** | ENABLE_MD5_SIGNED_CERTS disabled by default |

## 2.2 Selecting Cipher Suites

The strength of the secure communications is 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 or ECDH). Currently, when Diffie-Hellman is chosen it is used almost exclusively in ephemeral mode (DHE or ECDHE) in which new private key pairs are generated for each connection to allow perfect forward secrecy. The trade-off 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 elliptic curve variations on the Diffie-Hellman algorithms are denoted ECDH or ECDHE in the cipher suite name.

Authentication algorithms specify 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 key 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 and SHA256 recommended for new implementations. 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 |
| TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256 | ECDH | ECDSA | AES-128 | SHA-256 |

The AES_GCM cipher combines the encryption and digest hash components into a single algorithm so a dedicated hash algorithm is not used in these suites.

### Symmetric Algorithms Supported by MatrixSSL

| Algorithm | Recommended? | Typical Risks |
|---|---|---|
| RC4 | No | Several known weaknesses. Can be OK for small amounts of data. |
| 3DES | No | Theoretical weaknesses. AES typically a better candidate. |
| SEED | No | Standard usage only in Korea. AES is a better candidate. |
| AES | Yes | AES-256 preferred over AES-192 and AES-128. Lucky13 Attack mitigated internally. |
| AES-GCM | Yes | AES-256 preferred over AES-192 and AES-128. Lucky13 Attack mitigated internally. Without hardware acceleration, can be slower than AES-SHA. Risk that an as-yet undiscovered AES attack will compromise both encryption and record validation. |

### Hash Algorithms Supported by MatrixSSL

| Algorithm | Recommended? | Typical Risks |
|---|---|---|
| MD2 | No | Known weak. Used only for legacy certificate signatures. USE_MD2 disabled by default. |
| MD4 | No | Known weak. Used only for legacy certificate signatures. USE_MD4 disabled by default. |
| MD5 | No | Proven attacks. SSL 3.0 through TLS 1.1 require MD5 in combination with SHA-1 for their internal protocol (and therefore are at least as strong as SHA-1). TLS 1.2 does not require MD5. All MD5 based cipher suites disabled by default. ENABLE_MD5_SIGNED_CERTS disabled by default. |

| | | |
|---|---|---|
| SHA-1 | Yes | SHA-1 is widely deployed despite recent collision attacks. Only TLS 1.2 and newly issued certificates using SHA-2 are able to remove SHA-1 completely from the TLS protocol. |
| SHA-256 | Yes | Assumed secure. |
| SHA-384 | Yes | Assumed secure. |
| SHA-512 | Yes | Assumed secure. |

## Key Exchange Algorithms Supported by MatrixSSL

| Algorithm | Key Size | Recommended? | Typical Risks |
|---|---|---|---|
| RSA | **< 1024** | **No** | Weak. Below MIN_RSA_SIZE connections will be refused. |
| RSA | **1024** | **No** | In wide usage. Recommended to not use going forward |
| RSA | **> 1024** | **Yes** | Recommend at least 2048 bit keys. |
| DH | **< 1024** | **No** | Weak. Below MIN_DH_SIZE connections will be refused. |
| DH | **1024** | **No** | In wide usage. Recommended to not use going forward |
| DH | **> 1024** | **Yes** | Recommend at least 2048 bit DH group. |
| DHE/ECDHE | **1024 / 192** | **Yes** | See chart below. Ephemeral cipher suites provide perfect forward secrecy, and are generally the strongest available, although they are also the slowest performing for key exchange. |
| ECC | **>= 192** | **Yes** | 192 bit DH group and above is currently assumed secure. Smaller groups are not supported in MatrixSSL. Below MIN_ECC_SIZE connections will be refused. |
| PSK | **>= 128** | **Yes*** | Pre-shared Key ciphers rely on offline key agreement. *They avoid any weaknesses of Key Exchange Algorithms, however, it is not easy to change keys once they are installed when used as session keys. When PSK is used only for authentication (DHE_PSK cipher suites), the session encryption keys are generated each conncection. |

## Authentication Methods Supported by MatrixSSL

| Suite Type | Auth | Exchange | Recommended? | Typical Risks |
|---|---|---|---|---|
| RSA_WITH_NULL | **RSA** | **-** | **No** | No encryption. Authentication via RSA. Typically used for debugging connections only (since |
| DH_anon | **-** | **Diffie-Hellman** | **Yes*** | No Authentication. Key exchange only. *If used, authentication MUST be done by direct comparison of remote DH key ID to trusted key ID, similar to SSH authentication. If DH key ID authentication is done, this is similar in strength to DHE_PSK ciphers, although the keys exchanged are not ephemeral. Authentication to a trusted key ID can mitigate many attacks related to X.509 PKI infrastructure. |
| PSK | **Pre-shared Key** | **Pre-shared Key** | **Yes** | Pre-shared Keys can be used for authentication, since the same secret must be shared between client and server. DHE_PSK suites use PSK only for authentication, while PSK_ suites use PSK for authentication and session keys. PSK keys are difficult to change in the field, however authentication with PSK can mitigate many attacks related to X.509 PKI infrastructure. |
| RSA | **RSA** | **RSA** | **Yes** | The most commonly used authentication method. Supported by X.509 PKI infrastructure. Additional security can be had by directly comparing RSA key IDs to trusted Key Ids (similar to Certificate Pinning). Usually faster than ECC based authentication. |
| DHE_RSA | **RSA** | **Diffie-Hellman Ephemeral** | **Yes** | RSA for authentication, Ephemeral DH for key exchange. Provides Perfect forward secrecy. http://en.wikipedia.org/wiki/Forward_secrecy |

| | | | | |
|---|---|---|---|---|
| DHE_PSK | **PSK** | **Diffie-Hellman Ephemeral** | Yes | PSK for authentication, Ephemeral DH for key exchange. Does not rely on X.509. |
| ECDH_ECDSA | **ECC DSA** | **ECC Diffie-Hellman** | Yes | ECC DSA for authentication, ECC for key exchange. Most commonly used in embedded devices supporting hardware based ECC support. |
| ECDH_RSA | **RSA** | **ECC Diffie-Hellman** | Yes | ECC key exchange, with RSA authentication. Uses widely deployed X.509 RSA certificate infrastructure, but ECC for key exchange. Not often deployed due to the implementation having to support ECC and RSA. |
| ECDHE_ECDSA | **ECC DSA** | **ECC Diffie-Hellman Ephemeral** | Yes | ECC key exchange with ephemeral keys, ECC DSA authentication. Most commonly used in embedded devices supporting hardware based ECC support. |
| ECDHE_RSA | **RSA** | **ECC Diffie-Hellman Ephemeral** | Yes | Ephemeral counterpart to ECDH_RSA. |

## 2.3 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 as described in the next section.

Client authentication is also done inherently in Pre-shared Key cipher suites, as both sides of a connection must have a common shared secret.

## 2.4 Authentication and Key Exchange

### 2.4.1 Server and Client Authentication

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.

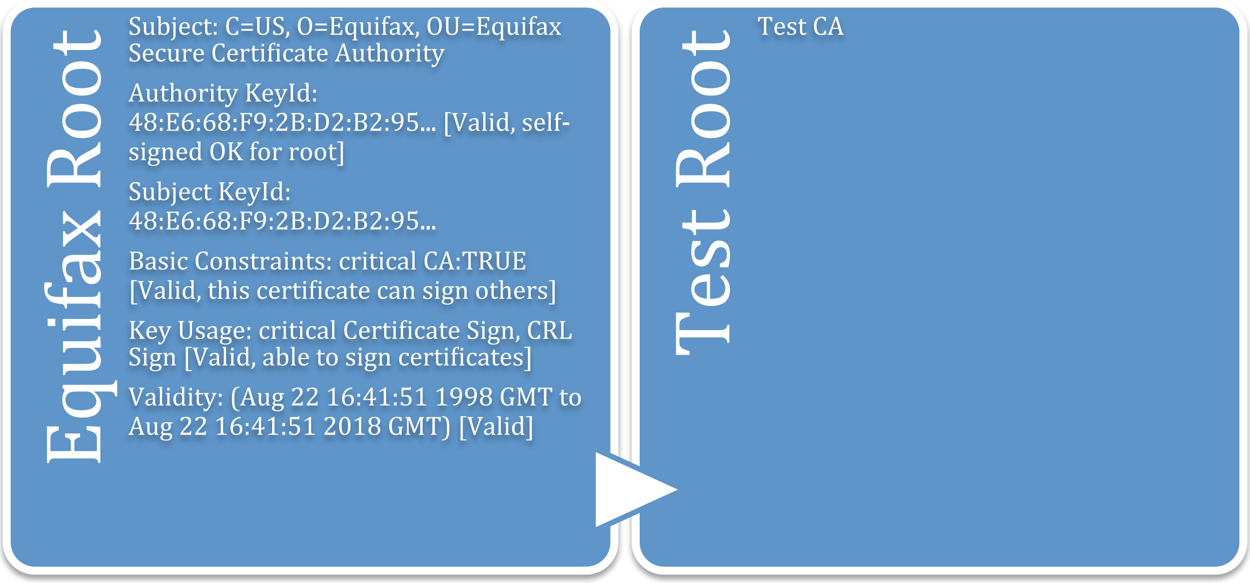
If client authentication is needed the mirror image of CA, certificate, and private key files must also be used. This chart shows which files clients and server must load when using a standard RSA based cipher suite such as `SSL_RSA_WITH_3DES_EDE_CBC_SHA`.

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

### 2.4.2 Certificate Validation and Authentication

Authentication in SSL is most often based on X.509 Certificate chain validation.

Example list of trusted root certificates loaded by a MatrixSSL client with `matrixSslLoadRsaKeys`.
https://www.geotrust.com/resources/root-certificates/



**Equifax Root**

Subject: C=US, O=Equifax, OU=Equifax Secure Certificate Authority

Authority KeyId: 48:E6:68:F9:2B:D2:B2:95... [Valid, self-signed OK for root]

Subject KeyId: 48:E6:68:F9:2B:D2:B2:95...

Basic Constraints: critical CA:TRUE [Valid, this certificate can sign others]

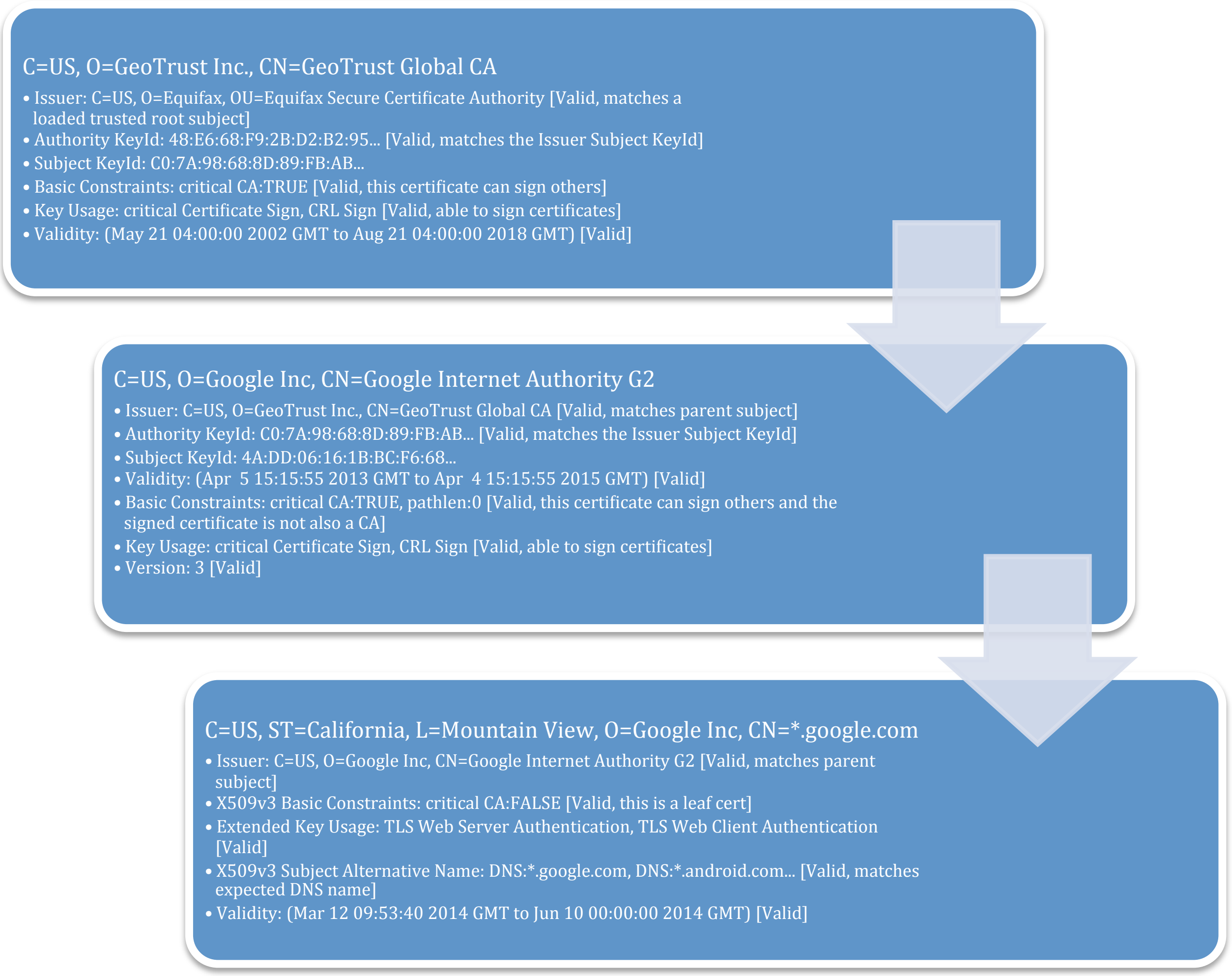
Key Usage: critical Certificate Sign, CRL Sign [Valid, able to sign certificates]

Validity: (Aug 22 16:41:51 1998 GMT to Aug 22 16:41:51 2018 GMT) [Valid]

**Test Root**

Test CA

Certificate chain sent to a MatrixSSL client during SSL handshake Certificate message by remote server www.google.com.



C=US, O=GeoTrust Inc., CN=GeoTrust Global CA

- Issuer: C=US, O=Equifax, OU=Equifax Secure Certificate Authority [Valid, matches a loaded trusted root subject]
- Authority KeyId: 48:E6:68:F9:2B:D2:B2:95... [Valid, matches the Issuer Subject KeyId]
- Subject KeyId: C0:7A:98:68:8D:89:FB:AB...
- Basic Constraints: critical CA:TRUE [Valid, this certificate can sign others]
- Key Usage: critical Certificate Sign, CRL Sign [Valid, able to sign certificates]
- Validity: (May 21 04:00:00 2002 GMT to Aug 21 04:00:00 2018 GMT) [Valid]

C=US, O=Google Inc, CN=Google Internet Authority G2

- Issuer: C=US, O=GeoTrust Inc., CN=GeoTrust Global CA [Valid, matches parent subject]
- Authority KeyId: C0:7A:98:68:8D:89:FB:AB... [Valid, matches the Issuer Subject KeyId]
- Subject KeyId: 4A:DD:06:16:1B:BC:F6:68...
- Validity: (Apr 5 15:15:55 2013 GMT to Apr 4 15:15:55 2015 GMT) [Valid]
- Basic Constraints: critical CA:TRUE, pathlen:0 [Valid, this certificate can sign others and the signed certificate is not also a CA]
- Key Usage: critical Certificate Sign, CRL Sign [Valid, able to sign certificates]
- Version: 3 [Valid]

C=US, ST=California, L=Mountain View, O=Google Inc, CN=*.google.com

- Issuer: C=US, O=Google Inc, CN=Google Internet Authority G2 [Valid, matches parent subject]
- X509v3 Basic Constraints: critical CA:FALSE [Valid, this is a leaf cert]
- Extended Key Usage: TLS Web Server Authentication, TLS Web Client Authentication [Valid]
- X509v3 Subject Alternative Name: DNS:*.google.com, DNS:*.android.com... [Valid, matches expected DNS name]
- Validity: (Mar 12 09:53:40 2014 GMT to Jun 10 00:00:00 2014 GMT) [Valid]

Checks that are done on all certificates:

| X.509 Field | New in 3.6 | Validation Performed |
|---|---|---|
| Version | New | Must be a version 3 certificate. Prior to 3.6, all certificate versions were accepted. |
| Serial | | Used for lookup in a CRL, if USE_CRL defined. |
| Signature Algorithm | | RSA or ECDSA algorithms. Must be SHA-1 or SHA-2 based hash. MD2 and MD5 support for RSA signatures is supported only with custom compile options. |
| Issuer | | In a chain, issuer must match the subject of the immediate (following) parent certificate. Self-signed certificates (Issuer == Subject) are allowed as loaded root certificates, but not as part of a chain. Common name must contain only printable characters. |
| Validity | Change | Current date must be within notBefore and notAfter range on all certs in the chain. Time is not currently validated. On platforms without a date function, the range check is always flagged as failed and must be handled by the Certificate Validation Callback. Prior to 3.6, example validation only within the Certificate Validation Callback. |
| Subject | New | Common name must contain only printable characters. Common name will be validated via full match to expectedName, if provided in matrixSslNewClientSession(). Partial match not allowed. Wildcard match is allowed for the first segment of a DNS name. Prior to 3.6, subject validation needed to be done within the Certificate Validation Callback. |
| Subject Public Key Info | | RSA and ECC keys supported. RSA public key modulus must be at least MIN_RSA_SIZE bits. ECC public key must be at least MIN_ECC_SIZE bits. |
| Signature | | The hash of the certificate contents must match the hash that is signed by the Issuer Public Key. |
| Basic Constraints | Change | For Root or intermediate certs, must be marked Critical with CA:TRUE. Path Length constraints are validated. Prior to 3.6, only CA flag was checked, not path length. |
| Key Usage | New | For Root or intermediate certs, must be marked for use as CertificateSign. For CRL checks, CrlSign flag must be set. Prior to 3.6, Key Usage was not enforced. |
| Extended Key Usage | New | If marked Critical, must have “TLS Web Server Authentication” or “TLS Web Client Authentication” set in the leaf certificate. Prior to 3.6, this extension was not enforced. |
| Subject Alternative Name | New | If an expectedName is specified in matrixSslNewClientSession() and does not match Subject Common Name, or any printable Subject Alternative Name of type DNS, Email or IP, validation will fail. Prior to 3.6, alt subject validation needed to be done within the Certificate Validation Callback. |
| Authority Key Identifier | New | If specified, the direct Issuer of the certificate must have a defined, matching Subject Key Identifier. Prior to 3.6 this field was ignored. |
| Subject Key Identifier | New | If specified, any direct children of the Issuer must have a defined, matching Authority Key Identifier. Prior to 3.6 this field was ignored. |
| CRL Distribution Points | | If USE_CRL is defined, matrixSslGetCRL() will download the CRL files from each URI type distribution point provided for each trusted root certificate (Note: not intermediate certificates). |
| CRL Validation | | CRL file must be signed by certificate with CrlSign Basic constraints. MD5 signatures not supported by default. |
| Unknown Extensions | Change | Unknown extensions are ignored, unless flagged as Critical. Validation will fail for any Critical extension unrecognized by MatrixSSL. Prior to 3.6, unknown critical extensions were warned, but allowed. |

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

# 3 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.

## 3.1 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.

## 3.2 Initialization

MatrixSSL must be initialized as part of the application initialization with a call to `matrixSslOpen`. This function 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 or EC 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`, `matrixSslLoadEcKeys`, `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 `matrixSslDeleteKeys`.

## 3.3 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 on a secure port. 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 all public APIs that act at a per-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 The Certificate Validation Callback Function section.

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

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 with `matrixSslNewSessionId` and pass it 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.

## 3.4 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 number 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 the 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 (or copies) 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. The return code from `matrixSslReceivedData` 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 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. |
| MATRIXSSL_REQUEST_RECV | Success. More data must be received and this function must be called again. User must first 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. |
| MATRIXSSL_RECEIVED_ALERT | Success. The data that was processed was an SSL alert message. In this case, the `ptbuf` 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 `ptbuf` and `ptLen` output parameters will be valid. The user may process the data directly from `ptbuf` or copy it aside for later processing. After handling the data the user must call `matrixSslProcessedData` to indicate the plain text data may be internally discarded |
| PS_SUCCESS | Success. This return code will be returned if the bytes parameter is 0 and there is no remaining internal data to process. This could be useful as a polling mechanism to confirm the internal buffer is empty. One real life use-case for this method of invocation is when dealing with a Google Chrome browser that uses False Start. |
| < 0 | Failure. See API documentation |

# 3.5 Communicating Securely With Peers

## 3.5.1 Encrypting Data

Once the handshake is complete, the application wishing to encrypt data that will be sent to the peer has the choice between two encoding options.

**In-Situ Encryption**

An in-situ encryption occurs when the outputted cipher text overwrites the plain text during the encoding process. In this case, the user will retrieve an allocated buffer from the MatrixSSL library, populate the buffer with the desired plaintext, and then notify the library that the plaintext is ready to be encoded. The API steps for the in-situ method are as follows:

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 number of bytes that were actually sent

**User provided plaintext data location**

The alternative to in-situ encryption is to allow the user to provide the location and length of the plaintext data that needs to be encoded. In this case, the encrypted data is still written to the internal MatrixSSL outdata buffer but the user provided plaintext data is left untouched. The API steps for this method are as follows:

1. The user passes the plaintext and length to `matrixSslEncodeToOutdata`
2. The application calls `matrixSslGetOutdata` to retrieve the encoded data and length to be sent (SSL always adds some overhead to the message size)
3. The application sends the out data buffer contents to the peer.
4. The application calls `matrixSslSentData` with the # of bytes that were actually sent

## 3.5.2 Decrypting Data

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 plain text
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.

## 3.6 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. Calling `matrixSslEncodeClosureAlert` and `matrixSslDeleteSession` will perform this step.

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` then `matrixSslSentData` mechanism that all outgoing data uses. Since the connection is being closed, the application shouldn't block indefinitely on sending the closure alert.

## 3.7 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.

# 4 CONFIGURABLE FEATURES

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

## 4.1 Protocol and Performance

| | |
|---|---|
| MATRIX_USE_FILE_SYSTEM | Define in the build environment. Enables file access for parsing X.509 certificates and private keys. |
| USE_CLIENT_SIDE_SSL | *matrixsslConfig.h* - Enables client side SSL support |
| USE_SERVER_SIDE_SSL | *matrixsslConfig.h* - Enables server side SSL support |
| USE_TLS | *matrixsslConfig.h* - Enables TLS 1.0 protocol support (SSL version 3.1) |
| USE_TLS_1_1 | *matrixsslConfig.h* - Enables TLS 1.1 (SSL version 3.2) protocol support. USE_TLS must be enabled |
| USE_TLS_1_2 | *matrixsslConfig.h* - Enables TLS 1.2 (SSL version 3.3) protocol support. USE_TLS_1_1 must be enabled |
| DISABLE_SSLV3 | *matrixsslConfig.h* - Disables SSL version 3.0. Defined by default since MatrixSSL 3.3.1. |
| DISABLE_TLS_1_0 | *matrixsslConfig.h* – Disables TLS 1.0 if USE_TLS is enabled but only later versions of the protocol are desired |
| DISABLE_TLS_1_1 | *matrixsslConfig.h* – Disables TLS 1.1 if USE_TLS_1_1 is enabled but only later versions of the protocol are desired |
| SSL_SESSION_TABLE_SIZE | *matrixsslConfig.h* – Applicable to servers only. The size of the session resumption table for caching session identifiers. Old entries will be overwritten when size is reached |
| SSL_SESSION_ENTRY_LIFE | *matrixsslConfig.h* – Applicable to servers only. The time in seconds that a session identifier will be valid in the session table. A value of 0 will disable SSL resumption |
| USE_STATELESS_SESSION_TICKETS | *matrixsslConfig.h* – Enable stateless session tickets as defined in RFC 5077 |
| ENABLE_SECURE_REHANDSHAKES | *matrixsslConfig.h* - Enable secure rehandshaking 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 |
| REQUESTED_MAX_PLAINTEXT_RECORD_LEN | *matrixsslConfig.h* – Enable the “max_fragment_length” TLS extension defined in RFC 4366. Value of #define determines fragment length (server may reject) |
| ENABLE_FALSE_START | *matrixsslConfig.h* – See code comments in file |

| | |
|---|---|
| USE_BEAST_WORKAROUND | *matrixsslConfig.h* – See code comments in file. |
| USE_CLIENT_AUTH | *matrixsslConfig.h* - Enables two-way(mutual) authentication |
| SERVER_CAN_SEND_EMPTY_CERT_REQUEST | *matrixsslConfig*.h – A client authentication feature. Allows the server to send an empty CertificateRequest message if no CA files have been loaded |
| SERVER_WILL_ACCEPT_EMPTY_CLIENT_CERT_MSG | *matrixsslConfig*.h – A client authentication feature. Allows the server to 'downgrade' a client authentication handshake to a standard handshake if client does not provide a certificate |
| USE_ZLIB_COMPRESSION | matrixsslConfig.h – Enables handshake support for zlib compression. See the section for zlib compression in this document for more information. |
| USE_PRIVATE_KEY_PARSING | *cryptoConfig*.h - Enables X.509 private key parsing |
| USE_PKCS5 | *cryptoConfig.h* - Enables the parsing of password protected private keys |
| USE_PKCS8 | *cryptoConfig.h* - Enables the parsing of PKCS#8 formatted private keys |
| USE_PKCS12 | *cryptoConfig.h* - Enables the parsing of PKCS#12 formatted certificate and key material |
| USE_1024_KEY_SPEED_OPTIMIZATIONS | *cryptoConfig.h* - Enables fast math for 1024-bit public key operations |
| PS_PUBKEY_OPTIMIZE_FOR_SMALLER_RAM<br>PS_PUBKEY_OPTIMIZE_FOR_FASTER_SPEED | *cryptoConfig.h* - RSA and Diffie-Hellman speed vs. runtime memory tradeoff. Default is to optimize for smaller RAM. |
| PS_AES_IMPROVE_PERF_INCREASE_CODESIZE<br>PS_3DES_IMPROVE_PERF_INCREASE_CODESIZE<br>PS_MD5_IMPROVE_PERF_INCREASE_CODESIZE<br>PS_SHA1_IMPROVE_PERF_INCREASE_CODESIZE | *cryptoConfig.h* - Optionally enable for selected algorithms to improve performance at the cost of increased binary code size. |
| ENABLE_MD5_SIGNED_CERTS | *cryptoConfig.h* – Support MD5 signature algorithm in X.509 certificates and Certificate Revocation Lists. |
| MIN_RSA_SIZE<br>MIN_DH_SIZE<br>MIN_ECC_SIZE | *cryptoConfig.h* – The minimum size in bits that MatrixSSL will accept for key exchange for each algorithm. Prevents weak keys from being used or downgraded to. |

## 4.2 Public Key Math Assembly Optimizations

Optimizing assembly code for low level math operations is available for many common processor architectures. The files *pstm_montgomery_reduce.c*, *pstm_mul_comba.c*, and *pstm_sqr_comba.c* in the *crypto/math* directory implement the available assembly optimizations. These following defines are set in the osdep.h header file by detecting the platform. These should be set accordingly when porting to an unsupported platform.

| | |
|---|---|
| PSTM_X86 | 32-bit x86 processor |
| PSTM_X86_64 | 64-bit x86 processor |
| PSTM_ARM | ARMv4 processor |
| PSTM_MIPS | 32 or 64 bit MIPS processor |
| PSTM_PPC | 32 bit PowerPC processor |
| <none of the above> | Standard C code implementation |

## 4.3 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 |

## 4.4 Minimum Firmware Configuration

MatrixSSL can be built to a minimum size using TLS 1.2, PSK cipher with AES128 and SHA256.

If interoperability with OpenSSL is desired, this standard cipher suite is not implemented (as of 1.0.1j). In this case, USE_SHA must also be defined and the cipher suite changed to USE_TLS_PSK_WITH_AES_256_CBC_SHA or USE_TLS_PSK_WITH_AES_128_CBC_SHA.

The MatrixSSL Performance Guide has more information on storage and memory requirements for various configurations and platforms.

To enable minimal configuration, all options in core/coreConfig.h, crypto/cryptoConfig.h and matrixssl/matrixsslConfig.h should be commented out, except for the following:

| | |
|---|---|
| *coreConfig.h* | Optional: USE_MATRIX_MEMORY_MANAGEMENT |
| *cryptoConfig.h* | USE_AES, USE_SHA256, USE_HMAC<br>Optional: __AES__ block to enable AESNI on Intel platforms.<br>Optional: For OpenSSL compatibility, also enable USE_SHA1 |
| *matrixsslConfig.h* | USE_TLS_PSK_WITH_AES_128_CBC_SHA256<br>USE_TLS, USE_TLS_1_1, USE_TLS_1_2, DISABLE_TLS_1_1, DISABLE_TLS_1_0, DISABLE_SSLV3<br>USE_CLIENT_SIDE_SSL and/or USE_SERVER_SIDE_SSL<br>Optional: SSL_DEFAULT_IN_BUF_SIZE, SSL_DEFAULT_OUT_BUF_SIZE set to 1500 for reduced RAM footprint.<br>Optional for Server: SSL_SESSION_TABLE_SIZE as low as 1 for reduced RAM footprint.<br>Optional: USE_DTLS<br>Optional: For OpenSSL compatibility, enable: USE_TLS_PSK_WITH_AES_256_CBC_SHA or USE_TLS_PSK_WITH_AES_128_CBC_SHA |

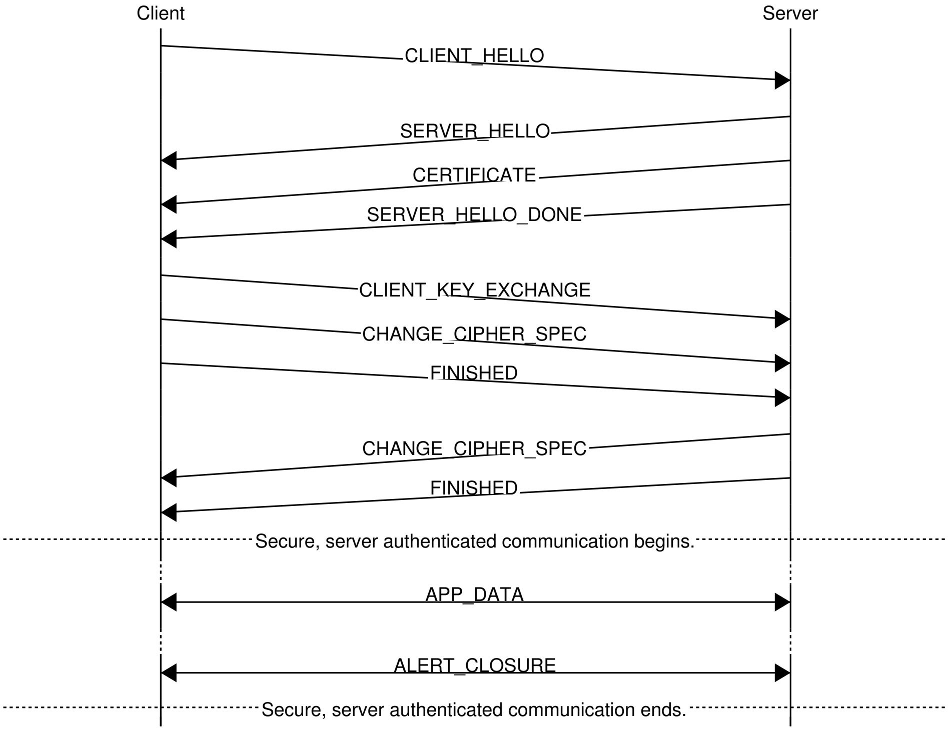
| **Code + Data Size** | **PSK_AES128_SHA256** | **PSK_AES128_SHA1** |
|---|---|---|
| ARM Thumb 2 | 24108 B | 25771 B |

# 5 SSL HANDSHAKING

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.

## 5.1 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
Secure, server authenticated communication begins.
APP_DATA
ALERT_CLOSURE
Secure, server authenticated communication ends.

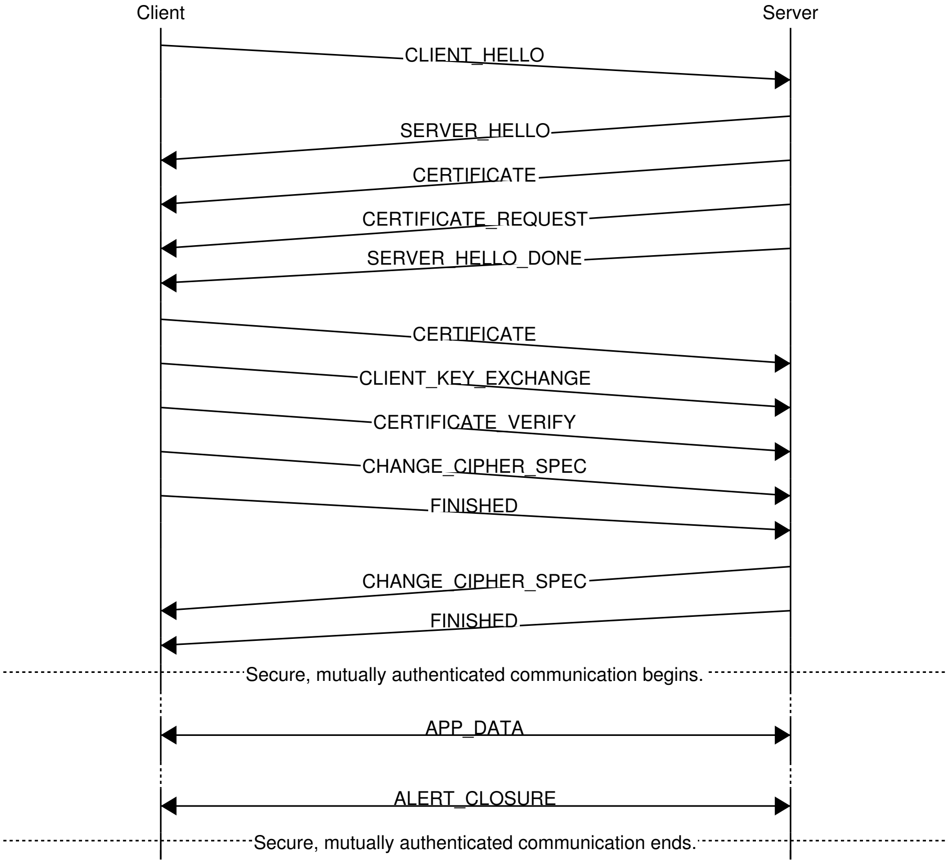
### Client Notes

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`.

## 5.2 Client Authentication Handshake

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
Secure, mutually authenticated communication begins.
APP_DATA
ALERT_CLOSURE
Secure, mutually authenticated communication ends.

**Client Notes**

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.

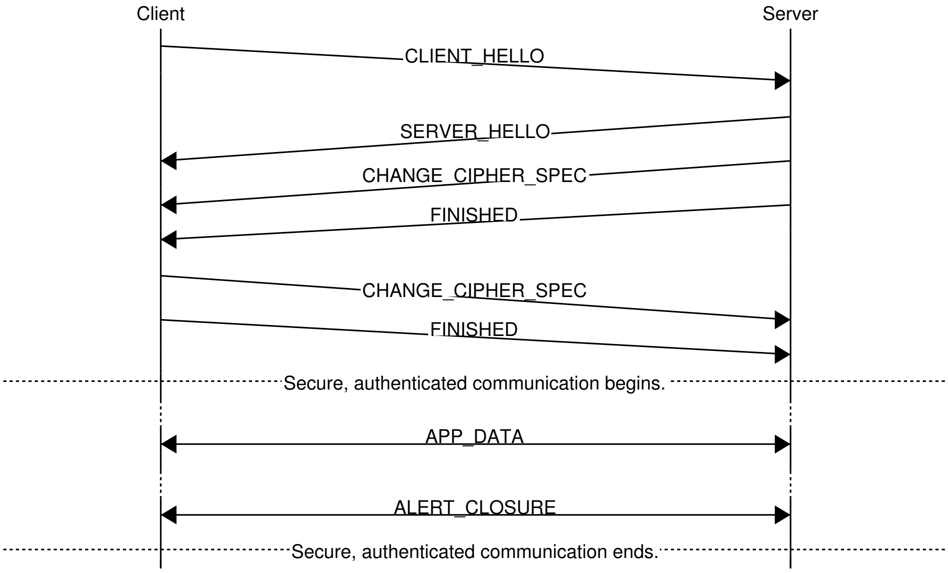
In order to participate in a client authentication handshake, the client must have loaded a Certificate Authority file during the call to `matrixSslLoadRsaKeys`.

**Server Notes**

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.

## 5.3 Session Resumption Handshake

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 Notes**

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`.

**Server Notes**

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`. Also, the server sends the Finished message first in this case, which is different from the standard handshake.

## 5.4 Other Handshakes

Other cipher suites can require variations on the handshake flights. PSK cipher suites do not use any key exchange. DSA cipher suites do not use certificates, and DH/DHE/ECDH/ECDHE cipher suites may or may not use certificates for authentication.

## 5.5 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`.

### 5.5.1 Disable Re-Handshaking At Runtime

Global disabling of re-handshakes can be controlled at compile time using the `ENABLE_SECURE_REHANDSHAKES` define but sometimes a per-session control of the feature is required. In these cases, the `matrixSslDisableRehandshakes` and `matrixSslReEnableRehandshakes` APIs are used.

### 5.5.2 The Re-Handshake Credit Mechanism

The re-handshake feature has been used at the entry point in a couple TLS attacks. In an effort to combat these attacks, MatrixSSL has incorporated a mechanism that prevents a peer from continually re-handshaking. This “re-handshake credit” mechanism is simply a count of how often the MatrixSSL-enabled application will allow a peer to request a re-handshake before sending the NO_RENEGOTIATION alert. The default number of credits is set using the `DEFAULT_RH_CREDITS` define in *matrixssllib.h*. The shipped default is 1.

In order to allow real-life conditions of re-handshakes, a single credit will be added after transmitting a given number of application data bytes. The default count of bytes that have to be sent before gaining a credit is set using the `BYTES_BEFORE_RH_CREDIT` define in *matrixssllib.h.* The shipped default is 20MB.

# 6 OPTIONAL FEATURES

This section describes some of the optional SSL handshake features. Additional details can be found in the API documentation for the specific functions that are referenced here.

## 6.1 Stateless Session Ticket Resumption

RFC 5077 defines an alternative method to the standard server-cached session ID mechanism. The stateless ticket mechanism allows the server to send an encrypted session ticket to the client that the client can use in a later connection to speed up the handshake process. The server does not have to store a large number of session ID entries when this stateless mechanism is used.

### Servers and Clients

The feature is made available with the `USE_STATELESS_SESSION_TICKETS` define in *matrixsslConfig.h*.

### Clients

Clients that wish to use the stateless session resumption mechanism must set the `ticketResumption` member of the `sslSessOpts_t` structure to `1` when calling `matrixSslNewClientSession`.

With that session option set, the client only has to use the standard session resumption API, `matrixSslNewSessionId`, to complete the use of the feature. If a server does not support stateless session tickets, the standard resumption mechanism will still work.

### Servers

The server must load at least one session ticket key using `matrixSslLoadSessionTicketKeys` to enable the feature. A user callback can optionally be registered that will be called each time a session ticket is received from a client. The callback will indicate to the user whether or not the server already has the correct ticket key cached. The callback can be used to locate a ticket key or to void the ticket and revert to a full handshake. The `matrixSslSetSessionTicketCallback` API is used to register this function.

The MatrixSSL implementation for resumption does not renew the session ticket as described in section 3.1 of the RFC (Figure 2). If the ticket is valid, the server progresses with the standard resumed handshake without a NewSessionTicket handshake message. If the server is unable to decrypt the session ticket, a full handshake will take place and a new session ticket will be issued. The MatrixSSL library also handles the expiration of a session ticket based on the value of the `SSL_SESSION_ENTRY_LIFE` in *matrixsslConfig.h*.

## 6.2 Server Name Indication Extension

RFC 6066 defines a TLS hello extension to allow the client to send the name of the server it is trying to securely connect to. This allows “virtual” servers to locate the correct server with the expected key material to complete the connection.

### Servers

Server applications should register the SNI callback using `matrixSslRegisterSNICallback`. This function must be called immediately after `matrixSslNewServerSession` before the first incoming flight from the client is processed. The callback will be invoked during the processing of the CLIENT_HELLO message if the client has included the SNI extension. The callback will use the incoming hostname to locate the correct key material and return them in the `sslKeys_t` structure format.

### Clients

Clients must include the SNI extension in the CLIENT_HELLO message. The utility function `matrixSslCreateSNIext` is provided to help format the extension given a hostname and hostname length. Once the extension format has been created it will be loaded into the `tlsExtension_t` structure with the `matrixSslLoadHelloExtension` API (`matrixSslNewHelloExtension` must first be called). The `tlsExtension_t` type is then passed to `matrixSslNewClientSession` to complete the client side SNI integration.

## 6.3 Maximum Fragment Length

RFC 6066 defines a TLS extension for negotiating a smaller maximum message size. The default maximum is 16KB (and can't be set larger). The only allowed sizes that may be negotiated are 512, 1024, 2048, or 4096 bytes. The client requests the feature in a CLIENT_HELLO extension and if the server agrees to the new maximum fragment length it will acknowledge that in the SERVER_HELLO reply.

### Clients

To request a smaller maximum fragment length the user sets the `maxFragLen` member of the `sslSessOpts_t *options` parameter to `512, 1024, 2048,` or `4096` when calling `matrixSslNewClientSession`. The server is free to deny the request.

### Servers

Servers will agree to the maximum fragment length request by default. To disable the feature for a session, the user may set the `maxFragLen` member of the `sslSessOpts_t *options` parameter to `-1` when calling `matrixSslNewServerSession`.

## 6.4 Truncated HMAC

RFC 6066 defines a TLS extension for negotiating an HMAC length of 10 bytes. The client requests the feature in a CLIENT_HELLO extension and if the server agrees to the truncation it will acknowledge that in the SERVER_HELLO reply.

### Clients

To request a truncated HMAC session the user sets the `truncHmac` member of the `sslSessOpts_t *options` parameter to `PS_TRUE` when calling `matrixSslNewClientSession`. The server is free to deny the request.

### Servers

Servers will agree to HMAC truncation by default. To disable the feature for a session, the user may set the `truncHmac` member of the `sslSessOpts_t *options` parameter to `-1` when calling `matrixSslNewServerSession`.

## 6.5 Application Layer Protocol Negotiation Extension

RFC 7301 defines a TLS hello extension that enables servers and client to agree on the protocol that will be used after the TLS handshake is complete. The idea is to embed the negotiation in the TLS handshake to save any round trips that might be needed to negotiate the protocol after the handshake. The extension works the same as any extension by the client sending a list of protocols it wishes to use in the CLIENT_HELLO and the server replying with an extension in the SERVER_HELLO. The trade-off for negotiating the protocol during the handshake is that both MatrixSSL servers and clients must be prepared to intervene in the middle of the handshake process via registered callback functions.

### Servers and Clients

The ALPN extension APIs will be available only if the `USE_ALPN` define in *matrixsslConfig.h* is enabled at compile-time. The define `MAX_PROTO_EXT` is the maximum number of protocols that can be expected in the list of protocols. The default is 8 and can be found in *matrixssllib.h*.

### Servers

Servers that wish to process ALPN extensions sent from a client must call the `matrixSslRegisterALPNCallback` function immediately after the session is created with `matrixSslNewServerSession`. The timing of the registration is important so that the callback can be associated with the proper session context before the first handshake message from the client is passed to `matrixSslReceivedData`.

The server ALPN callback that is registered with `matrixSslRegisterALPNCallback` must have a prototype of:

```
void ALPN_callback(void *ssl, short protoCount, char *proto[MAX_PROTO_EXT],
  int32 protoLen[MAX_PROTO_EXT], int32 *index)
```

The `ssl` parameter is the session context and may be typecast to an `ssl_t*` type if access is required.

The `protoCount` is the number of protocols that the client has sent in the CLIENT_HELLO extension. It is the count of the number of array entries in the `proto` and `protoLen` parameters to follow.

The `proto` parameter is the priority-ordered list of string protocol names the client wants to communicate with following the TLS handshake. The `protoLen` parameter holds the string lengths of the `proto` counterpart parameter for each protocol.

The `index` parameter is an **output** that the callback logic will assign based on the desired action:

- The index of the `proto` array member the server has agreed to use. **The index is the zero-based index to the array** so a return value of 0 will indicate the first protocol in the list. This selection will result in the server including its own ALPN extension in the SERVER_HELLO message with the chosen protocol.
- A negative value assigned to `index` indicates the server is not willing to communicate using any of the protocols. A fatal "no_application_protocol" alert will be sent to the client and the handshake will terminate.
- If the callback does not assign any value to the outgoing parameter, the server will not take any action. That is, neither a reply ALPN extension nor an alert will be sent to the client and the handshake will continue normally.

### Clients

To support this feature, clients must be able to generate the ALPN extension and also receive the server reply.

To generate the ALPN extension the API `matrixSslCreateALPNext` is used in conjunction with the `matrixSslNewHelloExtension`/`matrixSslLoadHelloExtension` framework. The `matrixSslCreateALPNext` API accepts an array of `unsigned char*` string values (array length of `MAX_PROTO_EXT`) along with a companion array that hold the string lengths for the protocol list. The function will format the protocols into the specified ALPN extension format and return that to the caller in the output parameters. Once the extension has been created the client must load the extension using the `matrixSslLoadHelloExtension` API (`matrixSslNewHelloExtension` must have been called as well). Finally, the extension must be passed to `matrixSslNewClientSession` in the extensions parameter. Here is what the ALPN extension creation and session start might look like:

```
tlsExtension_t      *extension;
unsigned char       *alpn[MAX_PROTO_EXT];
int32               alpnLen[MAX_PROTO_EXT];

matrixSslNewHelloExtension(&extension);

alpn[0] = psMalloc(NULL, strlen("http/1.0"));
```

```
memcpy(alpn[0], "http/1.0", strlen("http/1.0"));
alpnLen[0] = strlen("http/1.0");
alpn[1] = psMalloc(NULL, strlen("http/1.1"));
memcpy(alpn[1], "http/1.1", strlen("http/1.1"));
alpnLen[1] = strlen("http/1.1");

matrixSslCreateALPNext(NULL, 2, alpn, alpnLen, &ext, &extLen);
matrixSslLoadHelloExtension(extension, ext, extLen, EXT_ALPN);

psFree(alpn[0]);
psFree(alpn[1]);

matrixSslNewClientSession(&ssl, keys, sid, g_cipher, g_ciphers,
       certCb, g_ip, extension, extensionCb, &options);

matrixSslDeleteHelloExtension(extension);
```

To receive the server reply to the ALPN extension the client must register an extension callback routine using the `extCb` parameter when calling `matrixSslNewClientSession`. The callback will be invoked with the ALPN extension ID of `EXT_ALPN` (16) with a format of a single byte length followed by the protocol string value the server has agreed to.

See the example in *./apps/client.c* for full implementation details.

## 6.6 MatrixSSL Statistics Framework

Implementations that wish to capture counts of SSL events can tap into the MATRIXSSL_STATS framework by enabling USE_MATRIXSSL_STATS during the compile. The mechanism is a very simple counter that can be modified to record whatever specific SSL event the user wants. The default set of events capture the following:

- CLIENT_HELLO count (sent for clients and received for servers)
- SERVER_HELLO count (sent for servers and received for clients)
- Alerts sent
- Resumed handshake count
- Failed resumed handshake count
- Number of application data bytes received
- Number of application data bytes sent

To add an event to the framework the user must:

1. Add a member to the `matrixsslStats_t` data type in *matrixssllib.h*
2. Add a unique #define ID to the list of existing stats in *matrixssllib.h*
3. Add a handler for the new ID in the `matrixsslUpdateStat` function in *matrixssl.c*
4. Add a handler for the new ID in the `matrixsslGetStat` function in *matrixssl.c*
5. Add the call to `matrixsslUpdateStat` in the appropriate place in the MatrixSSL library

## 6.7 ZLIB Compression

The TLS specification specifies a mechanism for peers to agree on an algorithm to compress data before being encrypted. Although the feature is not widely adopted and is somewhat discouraged due to the 'CRIME' attack, there is limited support in MatrixSSL for zlib compression for implementations that are sensitive to throughput.

To enable the feature, enable `USE_ZLIB_COMPRESSION` in *matrixsslConfig.h*. It will also be necessary to edit the development environment to link with a zlib library. For a standard GCC POSIX environment this should simply mean including `-lz` in the linker flags.

The Matrix built-in support for this feature is limited. The feature only supports the internal compression and decompression of the FINISHED handshake message for initial handshakes. This means re-handshaking is not supported and that the application MUST compress and decompress application data manually.

**On the application data sending side:**

After a successful handshake with `USE_ZLIB_COMPRESSION` enabled, the user should call `matrixSslIsSessionCompressionOn` to test whether that mode has been successfully negotiated. If PS_TRUE, the user must manually zlib deflate any application data before calling the Matrix encryption functions. Do not compress more plaintext data in a single record than the maximum allowed record size to remain compatible with 3rd party SSL implementations.

**On the application data receiving side:**

Applications must test for `MATRIXSSL_APP_DATA_COMPRESSED` as the return code from `matrixSslReceivedData`. If found, the data must be zlib inflated to obtain the plaintext data