MatrixSSL Developer’s Guide

This document discusses developing with MatrixSSL. It includes instructions on integrating MatrixSSL into an application and a description of the configurable options for modifying the MatrixSSL library itself.

Integrating with applications

MatrixSSL is a library that provides a security layer to client and server applications allowing them to securely communicate with other SSL enabled entities. 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 client or server application the MatrixSSL library is being integrated with.

This document will walk through the specific points in which MatrixSSL should be integrated with an application. MatrixSSL APIs should be integrated into the application during initialization/cleanup, when new secure connections are being setup (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.

1. Initialization

MatrixSSL should 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 some cases the application may also call *matrixSslReadKeys* during its initialization. In server applications, this function takes the path to the certificate and private key files. In client applications, this function takes the trusted root certificate to validate server certificates. In either case, the call extracts the RSA material and returns an *sslKeys_t* structure to the application that will be used in a subsequent call to *matrixSslNewSession*. *matrixSslReadKeys* parses through an entire PEM certificate file so it is a rather CPU intensive task. This needs to be taken into consideration to determine the most logical place for your application to read in its keys. It can be called a single time at start up to keep the keys in memory for the life of the application. This is most useful if the application uses the same certificate file for each connection. Or it can be called 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. This API can also be used to read a set of keys for each virtual server, and reused for sessions within that virtual server in Apache like environments.

Once the application is done with the keys they are freed with a call to *matrixSslFreeKeys*.

2. Creating a Session

The next MatrixSSL integration point in the application is to identify when a new session is starting. In the case of a client, this is whenever it chooses to begin one since only client initiated sessions are supported in MatrixSSL. In the case of a server, a new session should be started when the server accepts a connection from a client. In a socket based application, this would typically happen when the *accept* socket call returns. The application sets up a new session with the API *matrixSslNewSession*. The returned *ssl_t* value will become the input parameter for most of the remaining APIs that act at a session level.

The required input parameters to *matrixSslNewSession* are the key structure from the previous call to *matrixSslReadKeys* and the flag *SSL_FLAGS_SERVER* or 0 (for a client session). For client cases, there is an optional *sessonId* parameter that identifies a previously open session to resume a session. A session id can be retrieved from a call to *matrixSslGetSessionId* once a session has been negotiated and before it is deleted. The session id parameter must always be NULL for server implementations.

A final client consideration at this point of integration is whether or not to register a certificate validation callback function with the *matrixSslSetCertValidator* API. This routine takes the SSL session and a function pointer as arguments. The registered function will be invoked during the portion of the handshake process in which the server’s certificate is being verified. This API should be used when the MatrixSSL default certificate validation is not deemed sufficient or if the client application would like to expose the certificate information to the user for any reason.

3. Handshaking

With the session established a client initiates a handshake by first constructing the CLIENT_HELLO message with a call to *matrixSslEncodeClientHello.* The client sends the constructed data to the server. When the server receives notice that a client is requesting a secure communication session and the function *matrixSslNewSession* has been called to create a new session structure the application can then read in the client message data. All incoming messages should be passed to *matrixSslDecode* which then processes the message and drives the handshake through the built-in SSLv3 state machine. The parameters to *matrixSslDecode* include the SSL structure returned from the call to *matrixSslNewSession*, input and output buffers, and alert and error output parameters. Refer to the API documentation for more details.

The *matrixSslDecode* API is a powerful function that processes handshake messages for clients and servers as well as decoding application data. Its return code tells the application what the message was and how it is to be handled.

The steps below outline the proper usage of *matrixSslDecode* on the server side when handshaking. A sockets based implementation is assumed in these steps.

1. The applications reads the client data off the socket with *recv*
2. *matrixSslDecode* is called with the client data
3. The return value from *matrixSslDecode* is tested to see what action the application is required to do with the output buffer. The list of possible return values and appropriate action include:
    a. SSL_SEND_RESPONSE - This value indicates the message was part of the SSLv3 standard and a reply is expected. The application should send the output buffer to the client with *send* and then call *matrixSslDecode* again to see if any more message data needs to be decoded.
    b. SSL_ERROR – This value indicates there has been an error while attempting to decode the data or that a bad message was sent. The application should attempt to send the out buffer to the client as a reply and then close the socket.
    c. SSL_ALERT – This value indicates the message was an alert sent from the client and the application should close the socket.
    d. SSL_PARTIAL – This value indicates that the input buffer was an incomplete message or record (or no data at all to parse). If the handshake is incomplete (*!matrixSslHandshakeIsComplete()),* the application must retrieve more data from the socket with *recv* and call *matrixSslDecode* with then entire record. If the handshake is complete, the caller can decide whether more data is expected at this point or not.
    e. SSL_FULL – This value indicates the output buffer was too small to hold the output message. The application should grow the output buffer and call *matrixSslDecode* again with the same input buffer. The maximum size of the buffer output buffer will never exceed 16K per the SSLv3 standard.
    f. SSL_PROCESS_DATA – This value indicates that the message is application specific data that does not require a response from the server. This message is an implicit indication that SSLv3 handshaking is complete. The decoded data has been written to the output buffer for application consumption.
    g. SSL_SUCCESS - A handshake message was successfully decoded and handled. No additional action is required for this message. *matrixSslDecode* can be called again immediately if more data is expected. This return code gives visibility into the handshake process and can be used in conjunction with *matrixSslHandshakeIsComplete* to determine when the handshake is complete and application data can be sent.

The client steps are identical to the server steps above except the process is started with a call to *matrixSslNewSession* and *matrixSslEncodeClientHello* to construct the first message to be sent to the server.

4. Communicating With Peers

Once the handshake is complete the application will simply wrap all incoming and outgoing messages with *matrixSslDecode* and *matrixSslEncode,* respectively.

5. 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 MatrixSSL 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.

On a graceful closure, a client application may wish to store aside the session id information before ending a session, to allow fast resumption of the next session to the same SSL server. It can do this with a call to *matrixSslGetSessionId* before calling *matrixSslDeleteSession*. Future negotiations with the same server can be quickly resumed by passing that session id to *matrixSslNewSession.*

6. Closing the library

At application exit the MatrixSSL library should be un-initialized with a call to *matrixSslClose*. If the application has called *matrixSslReadKeys* as part of the initialization process and kept its keys in memory it should call *matrixSslFreeKeys* before calling *matrixSslClose*. Additionally, if the client application has called *matrixSslGetSessionId* to support session resumption, it should call *matrixSslFreeSessionId* before calling *matrixSslClose*.

Working implementations of MatrixSSL integration can be seen in the *examples* subdirectory of the distribution package. A server source code example is available in the *httpsReflector* example application. A client source code example is available in the *httpsClient* example application.

Porting to Other Platforms

OS Dependent Code Layer

The code under *matrixssl/src/os* may need to be modified when porting to new platforms.

Build Environment Details

The supplied build environments allow the creation of a MatrixSSL shared object (DLL on Windows) library for each supported operating system. Details for Windows and

Linux builds are provided in this section. Use these examples as a guide for building on other platforms and make systems.

Windows Builds

The MatrixSSL package is distributed with Visual Studio .NET project files for use with Windows systems. If you are working with an earlier version of Visual Studio, the information in this section should be sufficient to create a MatrixSSL project.

Compiler and linker settings
Use the default compiler and linker settings for Debug and Release targets

Additional Debug defines
WIN32; _WIN32_WINNT=0x0500; _DEBUG; DEBUG

Additional Release defines
WIN32; _WIN32_WINNT=0x0500

Run-time library
*Multi-threaded DLL (*use Debug version for debug builds)

Linux Builds

The MatrixSSL package is distributed with *Makefile* files for building on Linux systems. These Makefiles can be used as templates for make systems on other platforms.

Debug defines
-DLINUX -DDEBUG

Release defines
-DLINUX

Debug compile options
-g

Release compile options
-O3

Extending MatrixSSL

This section of the developers guide explains more of the internals of MatrixSSL and how to extend its functionality.

Compile-Time Defines

A lot of the functionality of MatrixSSL has been encapsulated with compile-time definitions in the *matrixConfig.h* header file. Reducing the number of supported features is an effective way of reducing the compiled library size. Descriptions for these options can be found in the following list:

USE_SERVER_SIDE_SSL
On by default, this define enables server specific code to be compiled into the library. There is a small subset of public APIs that are only available to client side implementations. It is generally not advisable to disable server or client support in the MatrixSSL library unless the few Kb of savings is important to the project.

USE_CLIENT_SIDE_SSL
On by default, this define enables client specific code to be compiled into the library. There is a small subset of public APIs that are only available to client side implementations. It is generally not advisable to disable server or client support in the MatrixSSL library unless the few Kb of savings is important to the project.

USE_SSL_RSA_WITH_RC4_128_MD5
On by default, this define controls the inclusion of the cipher suite consisting of RSA public key encryption, 128 bit ARC4 symmetric encryption, and MD5 message authentication codes. This is the weakest cipher suite supported, and is marginally faster than the others.

USE_SSL_RSA_WITH_RC4_128_SHA
On by default, this define controls the inclusion of the cipher suite consisting of RSA public key encryption, 128 bit ARC4 symmetric encryption, and SHA1 message authentication codes. This ciphersuite is a good balance of speed and security for embedded devices.

USE_SSL_RSA_WITH_3DES_EDE_CBC_SHA
Off by default, this define controls the inclusion of the cipher suite consisting of RSA public key encryption, 128 bit 3DES symmetric encryption, and SHA1 message authentication codes. This is the strongest cipher suite in terms of security, but also the most CPU intensive.

USE_ENCRYPTED_PRIVATE_KEYS
On by default, this define controls whether or not to support the reading of 3DES encrypted (password protected) private key files passed to *matrixSslReadKeys*. Embedded MatrixSSL installations usually will not have an operator available to enter a password, so private keys are stored unencrypted on the device, and this option can be disabled to slightly reduce code size.

USE_MULTITHREADING
On by default, this define controls whether or not to use multithread mutex support in the operating system layer. This option can be enabled whether or not the library is used in a multithreaded application. It does not mean that MatrixSSL will generate any threads, it only provides additional concurrency control for environments that may have multiple SSL sessions in use simultaneously. This can be disabled in environments with no threading APIs defined in the MatrixSSL OS layer.

USE_PEERSEC_MALLOC
Off by default, this define controls whether or not to use the PeerSec memory routines for basic memory statistics. Should be turned off for release builds.

USE_FILE_SYSTEM
On by default, this define controls whether or not to include public API functions that make use of operating system calls that access files. The *matrixSslReadKeys* API is currently the only library function to use such system calls. A buffer only version of this API (*matrixSslReadKeysMem*) is included in the library.

Cipher Suites

MatrixSSL uses the term *cipher suite* to describe a collection of function callbacks and key size specifications used to determine which algorithms are used for symmetric and public key encryption/decryption and how MAC generation and verifications are handled for a session. The following section explains how each element of a cipher suite is implemented.

1. Defining a cipher suite
The list of available default cipher suites are found in the *supportedCiphers* static structure in the file *cipherSuite.c*. This structure defines all available cipher suites along with a required *NULL* suite as the last entry. The definition of the *sslCipherSpec_t* structure is as follows:

```
typedef struct {
      unsigned int      id;       // unique identifier
      unsigned char     macSize;  // MAC digest size (bytes)
      unsigned char     keySize;  // symmetric key length (bytes)
      unsigned char     ivSize; // symmetric block cipher iv size
      unsigned char     blockSize; // symmetric block cipher size
                                   // set to 1 for stream cipher
      //Init function
      int (*init)(sslSec_t *sec);
      //Cipher functions
      int (*encrypt)(sslCipherContext_t *ctx, char *in,
            char *out, int len);  // symmetric encryption
      int (*decrypt)(sslCipherContext_t *ctx, char *in,
            char *out, int len);  // symmetric decryption
      int (*encryptPub)(sslRsaKey_t *key, char *in, int inlen,
            char *out, int outlen);  // public key encryption
      int (*decryptPriv)(sslRsaKey_t *key, char *in, int inlen,
            char *out, int outlen);  // private key decryption
      int (*generateMac)(sslSec_t *sec, unsigned char type,
            char *data, int len, char *mac);
      int (*verifyMac)(sslSec_t *sec, unsigned char type,
            char *data, int len, char *mac);
} sslCipherSpec_t;
```

A cipher suite entry should be defined in the *supportedCiphers* structure between a custom define that has been added to the *matrixConfig.h* file. Any number of cipher suites can be compiled into the library. The SSL handshake protocol will negotiate the proper cipher at connection time. For reference, see the built-in supported cipher suites in the configuration header file:

USE_SSL_RSA_WITH_RC4_128_MD5
USE_SSL_RSA_WITH_RC4_128_SHA
USE_SSL_RSA_WITH_3DES_EDE_CBC_SHA

2. <u>Symmetric Encryption</u>
The symmetric encryption and decryption functions are identified in the *encrypt* and *decrypt* members of the *sslCipherSpec_t* structure. To add support for a new symmetric cipher context, locate the *sslCipherContext_t* structure definition in the header file of the chosen crypto provider and add the cipher context necessary to support the new method. For reference, see the PeerSec implementation in the header file *pscrypto.h*.

In addition to the callbacks, the values for *keySize*, *ivSize*, and *blockSize* all relate to symmetric encryption and should be set appropriately. The *keySize* member is the desired strength of the symmetric key in bytes. The *ivSize* is an optional length of an initialization vector if the chosen cipher requires one. The *blockSize* member should be specified if a block cipher is being used. If a stream cipher is used, set this value to 1.

3. <u>Public Key Encryption</u>
Public key encryption and decryption functions are identified in the *encryptPub* and *decryptPriv* members of the *sslCipherSpec_t* structure. In general, these callbacks are the least configurable members of a cipher suite. RSA is the standard in public key encryption and is assumed in the current MatrixSSL encryption layers. The value for *encryptPub* must be *matrixRsaEncryptPub* and the value for *decryptPriv* must be *matrixRsaDecryptPriv*. The implementation of these two functions will be implemented by a crypto provider.

4. <u>MAC Generation and Verification</u>
The message authentication code cipher is selected through the *generateMac* and *verifyMac* members of the *sslCipherSpec_t* structure. The MAC implementation is used during the handshake portion of negotiating a secure connection and is part of the SSLv3 specification. For this reason, it should not be necessary to replace the existing MatrixSSL MAC ciphers. For MD5 MACs, choose *md5GenerateMac* and *md5VerifyMac*, respectively. For SHA1 MACs, choose *sha1GenerateMac* and *sha1VerifyMac*.

Crypto Providers

MatrixSSL uses the term *crypto provider* to refer to the specific implementation of a cryptographic algorithm. In general, a crypto provider will implement an entire cipher suite, but it is possible that several crypto providers can contribute to a cipher suite. This allows the most appropriate version of a specific algorithm to be implemented for your application. The default crypto provider can be found in the MatrixSSL source code distribution in the *src/crypto* subdirectory. The set of functions a crypto provider must implement for the supplied cipher suites are prototyped in *cryptoLayer.h*. The following list provides details for each:

```
ARC4
matrixArc4Init(sslCipherContext_t *ctx, unsigned char *key, int keylen);
matrixArc4(sslCipherContext_t *ctx, char *in, char *out, int len);

3DES
matrix3desInit(sslCipherContext_t *ctx, const unsigned char *IV,
                    const unsigned char *key, int keylen);
matrix3desEncrypt(sslCipherContext_t *ctx, char *in, char *out, int len);
matrix3desDecrypt(sslCipherContext_t *ctx, char *in, char *out, int len);

MD5
matrixMd5Init(sslMd5Context_t *ctx);
matrixMd5Update(sslMd5Context_t *ctx, const unsigned char *buf,
                    unsigned long len);
matrixMd5Final(sslMd5Context_t *ctx, unsigned char *hash);

SHA1
matrixSha1Init(sslSha1Context_t *ctx);
matrixSha1Update(sslSha1Context_t *ctx, const unsigned char *buf,
                    unsigned long len);
matrixSha1Final(sslSha1Context_t *ctx, unsigned char *hash);

RSA
matrixRsaReadCert(char *fileName, unsigned char **out, int *outLen);
matrixRsaReadPrivKey(char *fileName, char *password, sslRsaKey_t **key);
matrixRsaFreeKey(sslRsaKey_t *key);
matrixRsaEncryptPub(sslRsaKey_t *key, char *in, int ilen, char *out, int outlen);
matrixRsaDecryptPriv(sslRsaKey_t *key, char *in, int ilen, char *out, int outlen);


X509 certificates
matrixX509ParseCert(unsigned char **certBuf, int certlen, sslRsaCert_t **cert);
matrixX509FreeCert(sslRsaCert_t *cert);
matrixX509ValidateCert(sslRsaCert_t *subjectCert, sslRsaCert_t *issuerCert);
matrixX509UserValidator(sslRsaCert_t *subjectCert,
                              int (*certValidator)(sslCertInfo_t *t));
```

Debugging

MatrixSSL provides the following debug functionality in *matrixConfig.h*:

sslAssert(C);

sslAssert is a macro defined in *matrixConfig.h* that allows a developer to test if a certain condition is true. In debug builds this will output a message containing the condition tested, the file name, and line number to stderr then calls sslBreak to stop the process. In release builds this simply outputs the message to stderr without breaking the process.

void sslBreak();
sslBreak allows a developer to stop the process and break into a debugger when an assert triggers. It is called by sslAssert in DEBUG builds.

void matrixStrDebugMsg(char *message, char *arg);
matrixStrDebugMsg is defined in *matrixSsl.c* and allows a developer to output a debug message to stdout with a single string argument (may be NULL). In release builds this function is compiled out.

void matrixIntDebugMsg(char *message, int arg);
matrixIntDebugMsg is defined in *matrixSsl.c* and allows a developer to output a debug message to stdout with a single integer argument. In release builds this function is compiled out.