Migrating to MatrixSSL 3

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 document is a reference for developers that are moving to MatrixSSL 3 from any previous version of the product.

Who Is This Document For?

- Software developers that are upgrading to MatrixSSL 3 from a previous version
- Anyone wanting to learn more about MatrixSSL 3

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 information in this document is relevant only to the commercially licensed version of MatrixSSL. Sections of this document that refer to the commercial version will be shaded.

Changes to the Public API

Changes By Functional Areas

These sections highlight the interface differences in 3 from a functionality perspective to help developers understand where they will need to implement upgrades.

Handshake Control and Application Data Exchange

The main change to the MatrixSSL interface in version 3 is to internalize management of the data buffers for the SSL handshake and application messages. MatrixSSL 3 still operates at a buffer interface to remain transport-independent only now the user if freed from having to manage the allocation and size of those buffers. MatrixSSL 3 now handles this automatically in the library.

A small set of pre MatrixSSL 3 functions including: `matrixSslDecode` and `matrixSslEncode` required the user to allocate and resize `sslBuf_t` data types for storing encoded and decoded SSL data. Return codes such as `SSL_FULL` and `SSL_PARTIAL` were used to indicate how the user should manage the buffers that stored the data.

This was a flexible design that worked well but **there were two common problems**:

1. The buffer management task was the most prone to user error during MatrixSSL integrations. The `matrixSslDecode` function could be particularly susceptible to misuse. The buffer related return codes of `SSL_FULL` and `SSL_PARTIAL` were conveyed using the same mechanism as the general success or failure of the function, so it required the user to be very diligent in handling all the numerous return cases. MatrixSSL 3 greatly simplifies this process and removes much of the burden of error checking from the user.

2. API misuse potentially wasted memory. APIs previous to MatrixSSL 3 did not return any size information back to the user, only buffer management codes. This resulted in users not being able to optimize the growth of the `sslBuf_t` types. The user was required to grow the buffer an unspecified amount and try to process the data again. In addition to the inefficiency of growing buffers without a specific target size to work from, many integrators chose to avoid the buffer resize issues altogether and simply allocated buffers to a maximum size from the start creating unused allocated memory. MatrixSSL 3 optimizes memory usage and frees developers from micromanaging buffers.

The buffer management changes introduced in version 3 solves these two problems through a new set of public interfaces that replace the `matrixSslEncode` and `matrixSslDecode` model.

The following table is the list of new APIs that control the handshaking and application data exchange. Complete function details can be found in the MatrixSSL API document.

Function Added in 3	Description
`matrixSslGetReadbuf`	Any time the application is expecting to receive data from a peer this function must be called to retrieve an allocated buffer that the incoming data should be read into.
`matrixSslGetOutdata`	Any time the application is expecting to send data to a peer this function must be called to retrieve the buffer containing the encoded SSL buffer
`matrixSslGetWritebuf`	This function is called when the user has application data that needs to be sent to the peer. This function will return an allocated buffer in which the user will copy the plaintext data that needs to be encoded and sent to the peer.
`matrixSslEncodeWritebuf`	This function is called when the user has application data that needs to be sent to the peer. This function will encrypt the plaintext data that has been copied into the buffer that was previously returned from a call to `matrixSslGetWritebuf`.
`matrixSslSentData`	This function must be called each time data has been sent to the peer.
`matrixSslReceivedData`	This function is used to indicate peer data has been copied into a buffer that was retrieved from `matrixSslGetReadbuf`. The decoding of data happens internally at this time and a plaintext application-layer record may be output to the user.
`matrixSslProcessedData`	This function is called after the user has finished processing plaintext application data that was returned from `matrixSslReceivedData`.

Client Session Initiation

MatrixSSL 3 now allows developers to initiate an SSL session with a single call to `matrixSslNewClientSession`. Previous versions required the client application to invoke `matrixSslNewSession`, `matrixSslSetCertValidator`, and `matrixSslEncodeClientHello`.

Server Session Initiation

Server-side session initiation now uses `matrixSslNewServerSession` rather than `matrixSslNewSession`.

Re-handshakes and Changing Session Options

MatrixSSL 3 has greatly simplified the re-handshake process and consolidated functionality into a single API call: `matrixSslEncodeRehandshake` which is available to both clients and servers. Previous versions required a number of different functions at various stages of the process. The functions that have been removed for use with re-handshaking are:

```
matrixSslSetCertValidator
matrixSslAssignNewKeys
matrixSslSetSessionOption
matrixSslEncodeHelloRequest
```

Loading Key Material

The `matrixSslReadKeys` function has been replaced by a mechanism in which a new key structure is first created and then specific key types are loaded into that structure. The new API set is `matrixSslNewKeys`, `matrixSslLoadRsaKeys` (`matrixSslLoadRsaKeysMem`), and `matrixSslDeleteKeys`.

Diffie-Hellman key loading uses the same model through the new `matrixSslLoadDhParams` function.

Client-side Session Resumption

The mechanism for clients to perform session resumption has been improved. The new mechanism is simply to pass a static `sslSessionId_t` structure pointer to `matrixSslNewClientSession` that will be populated with the session id when the connection is complete. Future calls to `matrixSslNewClientSession` can simply use the same structure pointer to provide the session id information for session resumption. The new API `matrixSslInitSessionId` will clear a `sslSessionId_t` structure. The functions `matrixSslGetSessionId` and `matrixSslFreeSessionId` have been removed.

Changes by API Name

This table shows how each public interface (or family of interfaces) has changed in MatrixSSL 3. If an API is not in this table, there was no change to the prototype or functionality.

Pre-3 API	3 API	Comments

`matrixSslReadKeys`
`matrixSslReadKeysMem`
`matrixSslFreeKeys`
	`matrixSslNewKeys`
	`matrixSslLoadRsaKeys`
	`matrixSslLoadRsaKeysMem`
	`matrixSslDeleteKeys`	

`matrixSslNewSession` `matrixSslSetCertValidator`	`matrixSslNewClientSession` `matrixSslNewServerSession`	
`matrixSslEncodeClientHello`	`matrixSslNewClientSession` `matrixSslEncodeRehandshake`	
`matrixSslEncodeHelloRequest`	`matrixSslEncodeRehandshake`	
`matrixSslAssignNewKeys` `matrixSslSetSessionOption`	`matrixSslEncodeRehandshake`	

`matrixSslDecode`
`matrixSslEncode`
`matrixSslHandshakeIsComplete`
	`matrixSslGetReadbuf`
	`matrixSslGetOutdata`
	`matrixSslSentData`
	`matrixSslReceivedData`
	`matrixSslProcessedData`
	`matrixSslGetWritebuf`
	`matrixSslEncodeWritebuf`
		The MatrixSSL Developer's Guide and the MatrixSSL API documents are the best source of information for these changes.

`matrixSslEncodeClosureAlert`	`matrixSslEncodeClosureAlert`	The prototype for this function has changed.
`matrixSslGetSessionId` `matrixSslFreeSessionId`	`matrixSslNewClientSession` `matrixSslInitSessionId`	

Code Examples

The following examples highlight the major architectural and API differences from a source code perspective between MatrixSSL version 3 and prior versions. This code is based on the PeerSec implementations of the HTTPS client and server applications that are provided in the MatrixSSL package. For purposes of brevity, a lot of error handling, code comments, and data definitions have been left out from these code snippets and, as such, they are not intended for copy-and-paste usage.

MatrixSSL APIs are highlighted in blue in the examples.

Client Session Initiation and Handshake Control

This code comparison for SSL handshake functionality illustrates the amount of buffer management surrounding `matrixSslDecode` was required in versions prior to 3. This is the functional area that has resulted in the greatest changes to the application layer and, in general, the developer will want to completely re-write the application layer for MatrixSSL 3 handshake functionality.

Code Example Prior to 3

```
int sslClientConnection(sslConn_t *cp, sslKeys_t *keys,
    sslSessionId_t *id, short cipherSuite,
    int (*certValidator)(sslCertInfo_t *t, void *arg))
{
  unsigned char buf[1024];

  matrixSslNewSession(&cp->ssl, keys, id, 0);
  matrixSslSetCertValidator(cp->ssl, certValidator, keys);

  cp->insock.size = cp->outsock.size = 1024;
  cp->insock.start = cp->insock.end = cp->insock.buf =
    (unsigned char *)malloc(cp->insock.size);
  cp->outsock.start = cp->outsock.end = cp->outsock.buf =
    (unsigned char *)malloc(cp->outsock.size);
  cp->inbuf.size = 0;
  cp->inbuf.start = cp->inbuf.end = cp->inbuf.buf = NULL;

  matrixSslEncodeClientHello(cp->ssl, & cp->outsock, cipherSuite);
  socketWrite(cp->fd, & cp->outsock); /* Platform send() */

READ_MORE:
  rc = sslRead(cp, buf, sizeof(buf));
  if (rc == 0) {
    if (matrixSslHandshakeIsComplete(cp->ssl) == 0) {
       goto READ_MORE;
    }
  } else if (rc > 0) {
    goto READ_MORE;
  } else {
    return -1; /* error */
  }
  return 0;
}
```


```
int sslRead(sslConn_t *cp, char *buf, int len)
{
  int bytes, rc, remaining;
  unsigned char   error, alertLevel, alertDescription, performRead;

  if (cp->inbuf.buf) {
    if (cp->inbuf.start < cp->inbuf.end) {
      remaining = (int)(cp->inbuf.end - cp->inbuf.start);
      bytes = (int)min(len, remaining);
      memcpy(buf, cp->inbuf.start, bytes);
      cp->inbuf.start += bytes;
      return bytes;
    }
    free(cp->inbuf.buf);
    cp->inbuf.buf = NULL;
  }
  if (cp->insock.buf < cp->insock.start) {
    if (cp->insock.start == cp->insock.end) {
      cp->insock.start = cp->insock.end = cp->insock.buf;
    } else {
      memmove(cp->insock.buf, cp->insock.start,
        cp->insock.end - cp->insock.start);
      cp->insock.end -= (cp->insock.start - cp->insock.buf);
      cp->insock.start = cp->insock.buf;
    }
  }
  /* Read up to as many bytes as there are remaining in the buffer */
  performRead = 0;

READ_MORE:
  if (cp->insock.end == cp->insock.start || performRead) {
    performRead = 1;
    bytes = socketWrite(cp->fd, cp->insock);
    cp->insock.end += bytes;
  }
  cp->inbuf.start = cp->inbuf.end = cp->inbuf.buf = malloc(len);
  cp->inbuf.size = len;

DECODE_MORE:
  error = 0;
  alertLevel = 0;
  alertDescription = 0;
```


```
rc = matrixSslDecode(cp->ssl, &cp->insock, &cp->inbuf, &error, &alertLevel,
  &alertDescription);
switch (rc) {
  case SSL_SUCCESS:
    return 0;

  case SSL_PROCESS_DATA:
    rc = (int)(cp->inbuf.end - cp->inbuf.start);
    rc = min(rc, len);
    memcpy(buf, cp->inbuf.start, rc);
    cp->inbuf.start += rc;
    return rc;

  case SSL_SEND_RESPONSE:
    bytes = socketWrite(cp->fd, cp->inbuf);
    cp->inbuf.start = cp->inbuf.end = cp->inbuf.buf;
    return 0;

  case SSL_ERROR:
    goto readError;

  case SSL_ALERT:
      goto readError;

  case SSL_PARTIAL:
    if (cp->insock.start == cp->insock.buf && cp->insock.end ==
        (cp->insock.buf + cp->insock.size)) {
      if (cp->insock.size > SSL_MAX_BUF_SIZE) {
        goto readError;
      }
      cp->insock.size *= 2;
      cp->insock.start = cp->insock.buf =
        (unsigned char *)realloc(cp->insock.buf, cp->insock.size);
      cp->insock.end = cp->insock.buf + (cp->insock.size / 2);
    }
    if (!performRead) {
      performRead = 1;
      free(cp->inbuf.buf);
      cp->inbuf.buf = NULL;
      goto READ_MORE;
    } else {
```

```
        goto readZero;
      }

    case SSL_FULL:
      cp->inbuf.size *= 2;
      if (cp->inbuf.buf != (unsigned char*)buf) {
        free(cp->inbuf.buf);
        cp->inbuf.buf = NULL;
      }
      cp->inbuf.start = cp->inbuf.end = cp->inbuf.buf =
        (unsigned char *)malloc(cp->inbuf.size);
      goto DECODE_MORE;
  }

 readZero:
  if (cp->inbuf.buf == (unsigned char*)buf) {
    cp->inbuf.buf = NULL;
  }
  return 0;
 readError:
  if (cp->inbuf.buf == (unsigned char*)buf) {
    cp->inbuf.buf = NULL;
  }
  return -1;
}
```

Code Example for version 3

```
int sslClientConnection(sslKeys_t *keys, sslSessionId_t *sid, SOCKET fd)
{
  int             rc, transferred, len, complete;
  ssl_t           *ssl;
  unsigned char   *buf;

  matrixSslNewClientSession(&ssl, keys, sid, 0, certCb, NULL, NULL);

 WRITE_MORE:
  while ((len = matrixSslGetOutdata(ssl, &buf)) > 0) {
    transferred = socketWrite(fd, buf, len, 0);
    if ((rc = matrixSslSentData(ssl, transferred)) < 0) {
      goto L_CLOSE_ERR;
    }
    if (rc == MATRIXSSL_REQUEST_CLOSE) {
      closeConn(ssl, fd);
      return MATRIXSSL_SUCCESS;
    }
    if (rc == MATRIXSSL_HANDSHAKE_COMPLETE) {
      /* This occurs on a resumption handshake */
      if (httpWriteRequest(ssl) < 0) { /* application data write */
        goto L_CLOSE_ERR;
      }
      goto WRITE_MORE;
    }
    /* MATRIXSSL_REQUEST_SEND is handled by loop logic */
  }

 READ_MORE:
  if ((len = matrixSslGetReadbuf(ssl, &buf)) <= 0) {
    goto L_CLOSE_ERR;
  }
  transferred = socketRead(fd, buf, len, 0);
  if ((rc = matrixSslReceivedData(ssl, (int32)transferred, &buf,
      (uint32*)&len)) < 0) {
    goto L_CLOSE_ERR;
  }

  switch (rc) {
```


```
    case MATRIXSSL_HANDSHAKE_COMPLETE:
      /* We got the Finished SSL message, initiate the HTTP req */
      httpWriteRequest(ssl); /* write application data */
      goto WRITE_MORE;

    case MATRIXSSL_APP_DATA:
      httpProcessResponse(buf, len); /* received application data */
      matrixSslProcessedData(ssl);
      closeConn(ssl, fd);
      return MATRIXSSL_SUCCESS;

    case MATRIXSSL_REQUEST_SEND:
      goto WRITE_MORE;

    case MATRIXSSL_REQUEST_RECV:
      goto READ_MORE;

    case MATRIXSSL_RECEIVED_ALERT:
      matrixSslProcessedData(ssl);
      goto READ_MORE;

    default:
      goto L_CLOSE_ERR;
  }

 L_CLOSE_ERR:
  matrixSslDeleteSession(ssl);
  return MATRIXSSL_ERROR;
}
```

Encoding And Sending Application Data

Again, the first thing that will be obvious to the reader is how much buffer management overhead was required in previous versions.

Code Example Prior to 3

```
int sslWriteAppData(sslConn_t *cp, char *buf, int len, int *status)
{
  int             rc;

  *status = 0;

  if (cp->outsock.buf < cp->outsock.start) {
    if (cp->outsock.start == cp->outsock.end) {
      cp->outsock.start = cp->outsock.end = cp->outsock.buf;
    } else {
      memmove(cp->outsock.buf, cp->outsock.start,
        cp->outsock.end - cp->outsock.start);
      cp->outsock.end -= (cp->outsock.start - cp->outsock.buf);
      cp->outsock.start = cp->outsock.buf;
    }
  }
  /*
    If there is buffered output data, the caller must be trying to
    send the same amount of data as last time.  We don't support
    sending additional data until the original buffered request has
    been completely sent.
  */
  if (cp->outBufferCount > 0 && len != cp->outBufferCount) {
    return -1;
  }
  /* If we don't have buffered data, encode the caller's data */
  if (cp->outBufferCount == 0) {

 RETRY_ENCODE:
    rc = matrixSslEncode(cp->ssl, (unsigned char *)buf, len, &cp->outsock);
    switch (rc) {
      case SSL_ERROR:
        return -1;

      case SSL_FULL:
```

```
        if (cp->outsock.size > SSL_MAX_BUF_SIZE) {
          return -1;
        }
        cp->outsock.size *= 2;
        cp->outsock.buf =
          (unsigned char *)realloc(cp->outsock.buf, cp->outsock.size);
        cp->outsock.end =
          cp->outsock.buf + (cp->outsock.end - cp->outsock.start);
        cp->outsock.start = cp->outsock.buf;
        goto RETRY_ENCODE;
    }
  }
  /* We've got data to send */
  rc = send(cp->fd, (char *)cp->outsock.start,
    (int)(cp->outsock.end - cp->outsock.start), MSG_NOSIGNAL);
  if (rc == SOCKET_ERROR) {
    *status = getSocketError();
    return -1;
  }
  cp->outsock.start += rc;
  /*
    If we wrote it all return the length, otherwise remember the number of
    bytes passed in, and return 0 to be called again later.
  */
  if (cp->outsock.start == cp->outsock.end) {
    cp->outBufferCount = 0;
    return len;
  }
  cp->outBufferCount = len;
  return 0;
}
```

Code Example for version 3

```
int32 int sslWriteAppData(ssl_t *cp, char *data, int len, SOCKET fd)
{
  unsigned char   *buf;
  uint32          available;
  in32            rc;

  if ((available = matrixSslGetWritebuf(cp, &buf, len)) < 0) {
    return -1;
  }
  memcpy((char *)buf, data, available);
  if (matrixSslEncodeWritebuf(cp, strlen((char *)buf)) < 0) {
    return -1;
  }

WRITE_MORE:
  len = matrixSslGetOutdata(cp, &buf);
  transferred = send(fd, buf, len, 0);
  /* Indicate that we've written > 0 bytes of data */
  if ((rc = matrixSslSentData(cp, transferred)) < 0) {
    return -1;
  }
  if (rc == MATRIXSSL_REQUEST_SEND) {
    goto WRITE_MORE;
  }
  return 0;
}
```