Version 1.0

Robin Seggelmann (seggelmann@fh-muenster.de)

In this document the Datagram Transport Layer Security (DTLS) protocol, a modification of the Transmission Control Protocol (TCP) for unreliable transport protocols, and its extensions SCTP-aware DTLS and Heartbeats will be introduced. OpenSSL has currently the most advanced open source implementation, which will be described including its API.

Datagram Transport Layer Security

The Transport Layer Security (TLS) protocol [1] is a widely deployed security solution for reliable transport protocols. Although it has been developed for any transport protocol which is reliable and maintains the order of the messages, these requirements are only met without limitations by the Transmission Control Protocol (TCP). Securing the unreliable User Datagram Protocol (UDP) as well as the Stream Control Transmission Protocol (SCTP) [2] is not possible or only very limited. Therefore, TLS was modified to allow unreliable and out of order transfer, which resulted in Datagram Transport Layer Security (DTLS), as described in RFC 4347 [3].

Protocol Introduction

TLS has been designed for reliable transport protocols, that is it expects no lost or reordered messages from the transport layer, which not even has to be message-oriented, but can also be a simple byte stream. If it detects that a message is lost or out of order, it reasonably assumes an attack and drops the connection. Unfortunately, this makes it impossible to use it with unreliable transport protocols, with which losses and reordering are very likely. So the main problem is to tolerate unreliability and not creating any security issues while doing so.

Base Protocol

The DTLS protocol has, like TLS, a base protocol called *Record Layer*, and four subprotocols on top of it. These are the *Handshake*, the *ChangeCipherSpec* and the *Alert protocol* as well as the application data protocol.

Record Layer

The header of the *Record Layer* consists of the *Content Type*, that is which subprotocol it is carrying, the *Protocol Version* and its *Length*. It retains message limits, in case the transport layer does not. Each *Record* message has a unique sequence number, which is increased with every message sent. TLS

maintains this number implicit on both peers, that is it is not transmitted. Nonetheless, it is used for the calculation of the Hashed Message Authentication Code (HMAC), which is used to ensure the integrity of the message. If a received message does not have the expected sequence number, the hash cannot be verified and the connection is dropped. This behavior is counterproductive with unreliable transport protocols, so the DTLS *Record Header* has been extended. The sequence number is part of the header, as well as the epoch, which is increased with every successful handshake and also used for the hash calculation. The extended DTLS *Record Header* is illustrated in Figure 1.

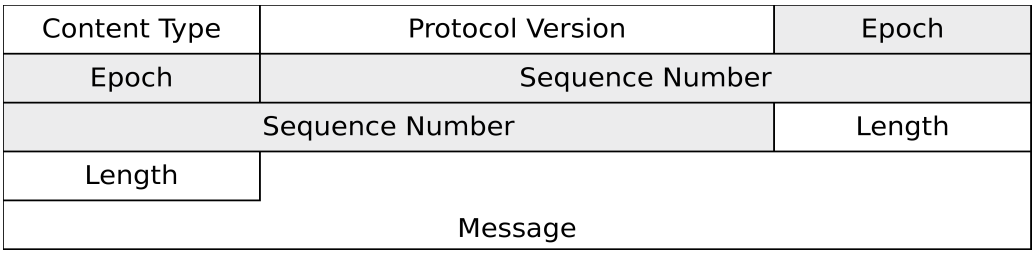

Figure 1: DTLS Record Header

Handshake Protocol

To set up a new connection and negotiate the security parameters, like cipher suite, hash algorithms or compression, the *Handshake protocol* is used. The client initiates the handshake by sending a *ClientHello* message to the server. This message contains the supported cipher suites, hash and compression algorithms and a random number. The server is supposed to respond with a *ServerHello*, which contains the cipher suite and algorithms the server has chosen from the ones the client offered and also a random number. Both random numbers will be used, among other data, to calculate the master secret. The server may continue with a *ServerCertificate* with its certificates to authenticate itself, if necessary. In that case it can also send a *CertificateRequest*, to provoke the client to authenticate, too. For some cipher suites additional data is necessary for the calculation of the secret, which can be send with a *ServerKeyExchange*. Since the last three messages mentioned are optional, a *ServerHelloDone* indicates when no more messages follow from the server.

This part of the handshake is a problem with connection-less transport protocols, because there is no transport connection setup necessary and an attacker could just send many *ClientHellos* to a server. This could be used for a Denial of Service (DOS) attack against the server, which will start a new session, thus allocating resources, for every *ClientHello*, or against another victim by redirecting the much larger response of the server to it, thus multiplying the attacker's bandwidth. To prevent this issue, DTLS uses an additional handshake message, called *HelloVerifyRequest*. It is sent in response to the *ClientHello* and contains a so-called cookie of arbitrary data, preferably signed. The server will only send this message without allocating any resources. The client then has to repeat its *ClientHello* with the cookie attached. If the cookie can be verified, hence the signature, the server knows that the client has not used a faked address, and since the *HelloVerifyRequest*

is small and has to be answered before the server sends any more data, no DOS attacks are possible anymore. After this verification the handshake will be continued as before, with the server finally sending the *ServerHello*.

After the *ServerHelloDone*, the client has to send its certificates with a *ClientCertificate* if the server requested authentication. This is followed by a *ClientKeyExchange*, which contains its public key or other cryptographic data, depending on the cipher suite used. Also depending on the cipher suite is whether a *CertificateVerify* to verify a signed certificate has to be sent. At this point both peers have enough information to calculate the master secret. Thus, the client sends the *ChangeCipherSpec*, to announce that the negotiated parameters and the secret will be used from now on. Its last message is the *Finished*, which contains a hash calculated over the entire handshake and is encrypted already. The server concludes the handshake by also sending the *ChangeCipherSpec* and *Finished*. The complete handshake sequence is depicted in Figure 2.

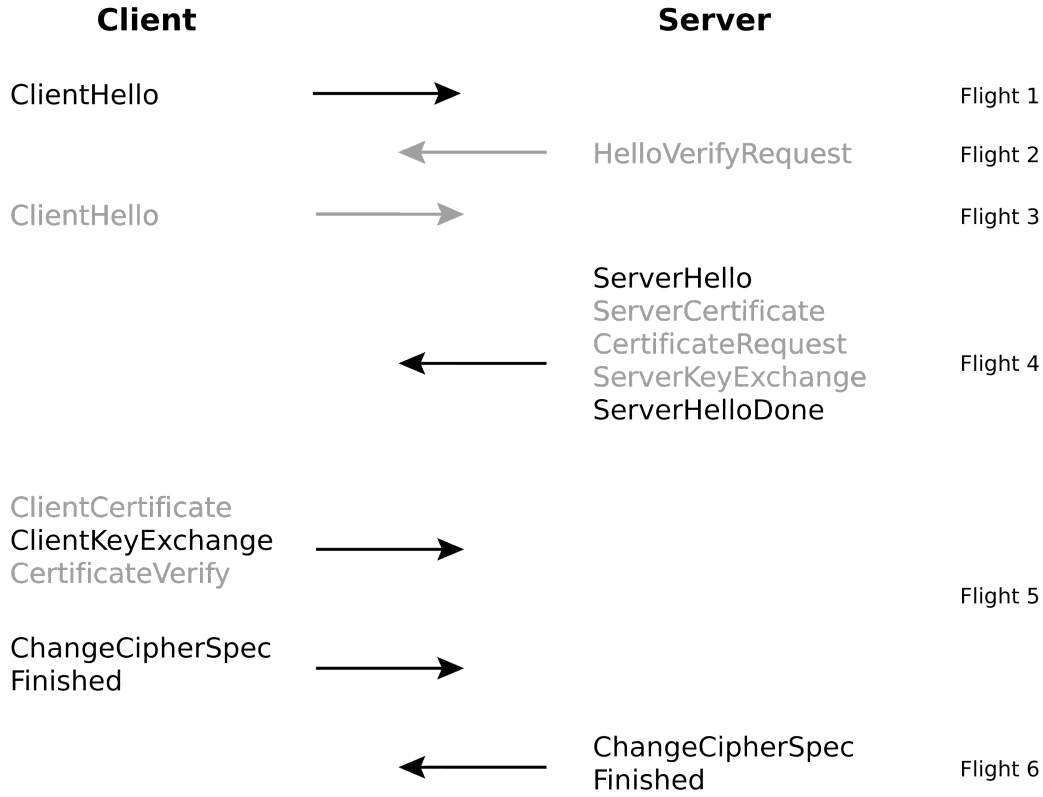

Figure 2: DTLS Handshake with Flights

Since the handshake cannot be completed if one or more messages are missing, it has to be performed reliable. With an unreliable transport, DTLS has to ensure the reliability of handshake messages itself. Therefore, it needs a timer to retransmit lost messages. For increased efficiency, DTLS does not use a timer for every message, but for bundles of messages, called flights. A flight contains all messages before the sending side changes (compare Figure 2). For every flight sent a timer is started, and if there is no response until the timer expires, the entire flight will be retransmitted.

Another issue is that most protocols other than TCP are message-oriented, while TCP is bytestream-oriented. TCP does not care how large a *Record* message is, it will just split it in as many parts as necessary to send it.

Message-oriented protocols on the other hand, may not have a mechanism to fragment and reassemble messages. The consequence is that only messages smaller than the current Path-MTU can be sent. The *Path Maximum Transmission Unit* is the smallest common message size every host on the path between the peers can handle. Especially the messages containing certificates may be larger than the current Path-MTU. To still be able to transfer these messages, DTLS has to provide its own fragmentation mechanism. This is achieved by extending the *Handshake Message Header*. With TLS every handshake message starts with its *Message Type* and its *Length*. For DTLS a *Fragment Offset* and *Fragment Length* entry is added.

DTLS also has to deal with reordered messages, which can likely occur with unreliable transports. To handle handshake messages arriving in the wrong order, the Handshake Message Header is further extended and a *Message Sequence Number* is added. This allows to restore the correct sequence of the handshake. The new header is shown in Figure 3.

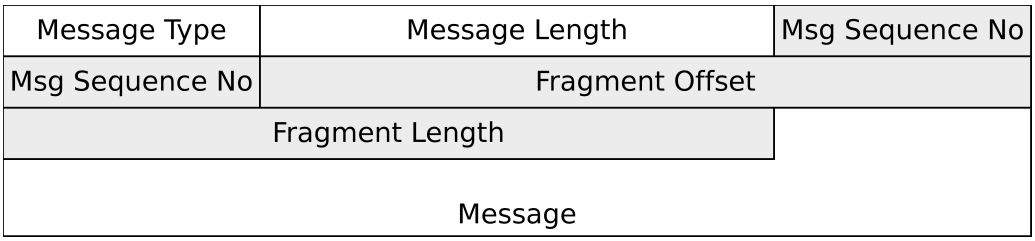

Figure 3: DTLS Handshake Message Header

Alert Protocol

The Alert protocol is used to notify warnings or errors that might have occurred, for example if a certificate could not be verified. While errors are always fatal and lead to the immediate shutdown of the connection, warnings are informational and the connection can remain established. With DTLS some errors are just sent as warnings, like *BadRecordMAC*, *RecordOverflow* or *DecryptionFailed*, because otherwise the connection-less protocol would allow an attacker to shut down the connection with an arbitrary message to one of the peers. Additionally, alert messages are also used to gracefully shut down the connection. When a peer has nothing to send anymore, it should send a *CloseNotify* alert. The connection is closed after both peers have sent it.

Replay Check

An attacker is unable to modify messages, due to the encryption and integrity checks. However, with a connection-less transport protocol he could just copy a valid message and resend it to the receiver. If he has knowledge of the application protocol used, he may be able to reissue a command in this way. To prevent that, DTLS has its own *Replay Check*. A window is maintained in which *Record Sequence Numbers* of received messages are valid, if not already seen. Every other message will be dropped.

Heartbeat Extension

When using connection-less transport protocols, there is no acknowledgement of received data, so when the receiver does not return any data, a sender does not know if it is still alive. If the used application protocol does not provide any mechanism to check if the peer still exists and responds, the only way is to initiate a handshake for renegotiation, which is quite inconvenient.

The *Heartbeat Extension* for TLS and DTLS [4] adds two new messages to the protocol, the *HeartbeatRequest* and the *HeartbeatResponse*. These can be used to realize a keep-alive functionality, because every received *HeartbeatRequest* has to be responded with a *HeartbeatResponse* immediately. Both messages consist of their type, length, an arbitrary payload and padding, as shown in Figure 4. The response to a request must always return the same payload but no padding. This allows to realize a Path-MTU Discovery by sending requests with increasing padding until there is no answer anymore, because one of the hosts on the path cannot handle the message size any more. The smaller response ensures that only one direction of the path is measured, because the routing and so the Path-MTU can be different on each way.

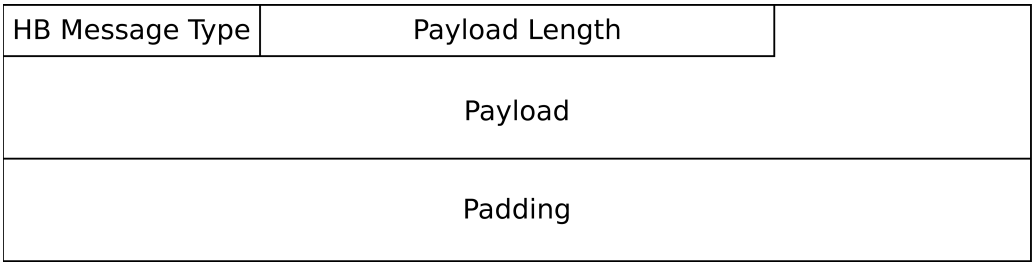

Figure 4: Heartbeat Message Scheme

For backward compatibility, the *Heartbeat Extension* can only be used if both peers support it, otherwise the connection may be dropped with an *unexpected message* alert. This is achieved by adding so called *Hello Extensions* to the *ClientHello* and *ServerHello*, respectively. If both peers indicate their support, the extension can be used. Since mobile clients usually want to avoid any unnecessary traffic to save battery power, the *Hello Extension* can also be used to indicate if the host is actually willing to respond to Heartbeats or does not want to receive any requests at all, but preserves the possibility to send them itself.

SCTP-aware DTLS

DTLS can basically be used with SCTP since it has no requirements for the transport protocol, other than the transport itself. However, DTLS may drop messages in some scenarios, which is not appropriate with the reliable transport of SCTP and features like retransmission timers would be existing twice. Hence, some adaptations [5] are necessary to make DTLS SCTP-aware, as described in RFC 6083 [6].

Being developed for unreliable transport protocols, the default behavior of DTLS is discarding unexpected messages. This may occur when messages

arrive after a renegotiation has been performed and thus the key material has been changed. These messages cannot be decrypted anymore and will be dropped. SCTP supports multiple streams, that is multiple unidirectional channels within the same connection. This can be used to separate logically independent data from each other, for example retrieving each object of a website (text, images, videos, etc.) on a different stream. So the order of the messages has to be maintained per stream only, not across multiple streams. If a message is lost, only the messages of the same stream have to be delayed until the retransmission arrives, while without multi-streaming all messages will be delayed. Therefore, data across multiple streams is likely to arrive reordered. Additionally, SCTP supports unordered delivery within a stream as well. To prevent message loss, it has to be ensured that the data transmission is stopped and every outstanding message has been received before initiating a renegotiation. The Extension PR-SCTP to send messages unreliable is supported without any modifications, since DTLS is particularly made for unreliable protocols.

Since SCTP provides reliable transfer, DTLS' reliability mechanisms for the handshake are not necessary anymore and the timer and fragmentation must not be used. The replay check is done by SCTP as well, so this is also not necessary.

DTLS Implementation of OpenSSL

A prototype implementation of DTLS for OpenSSL was developed while it was specified and standardized in 2005 [7]. It is part of the official releases since version 0.9.8, but did not receive much attention until the release of version 1.0.0, which already contained many bug fixes [8]. However, the architecture and API of OpenSSL was designed for TLS and its TCP connections, which caused some difficulties and limitations when implementing DTLS.

The architecture of OpenSSL is basically split into three parts, the context (CTX), the session (SSL) and basic I/O functionality (BIO). The context knows which protocol, that is SSL version 2 or 3, TLS or DTLS, is used, holds a session cache and other global parameters. For every new session an SSL object is created from the context and uses these parameters. The SSL object itself holds the session state and a BIO object for I/O abstraction. The BIO object can communicate with a networking socket or another BIO object, creating a so-called chain of BIO objects. A possible combination could be a buffering BIO before the actual socket BIO.

When initializing the context with a protocol, an SSL_METHOD object will be assigned to it. This object is specific to the protocol and contains a set of functions for every action, like sending, receiving, handshaking and so on. Each SSL object created for a new connection with this context will map the generic API to these functions. Hence, the DTLS implementation was added with a DTLS specific SSL_METHOD and corresponding functions. This can already be used with the existing BIO objects, which are TCP specific though. To use another protocol, like UDP or SCTP, new BIO objects aware of their characteristics had to be created.

Since this architecture has been created for TCP based connections, the relation between SSL and socket BIO objects is always one-to-one. This results in the limitation that transport protocol connections can also only be one-to-one, although SCTP can be used and UDP is only used one-to-many style, that is handling multiple connections per socket. The one-to-many style cannot be realized with OpenSSL without elaborate modifications to its architecture and API, because multiple SSL objects would have to share a single BIO object. As a workaround, SCTP can only be used one-to-one style, like TCP, and UDP has to use connected sockets to simulate a one-to-one behavior.

OpenSSL DTLS API

The API used for DTLS is mostly the same as for TLS, because of the mapping of generic functions to protocol specific ones. Some additional functions are still necessary, because of the new BIO objects and the timer handling for handshake messages. The generic concept of the API is described in the following sections. Examples of applications using DTLS are available at [9].

Prerequisites

Every program using OpenSSL has to start with initializing the library by calling

```
SSL_load_error_strings();  /* readable error messages */
SSL_library_init();        /* initialize library */
```

before any other action can be done. The DTLS specific context can be created thereafter, from which SSL objects for each connection can be derived. The context is different for the client and server, and several parameters, including certificates and keys, have to be set:

```
/***** SERVER *****/
ctx = SSL_CTX_new(DTLSv1_server_method());

/***** CLIENT *****/
ctx = SSL_CTX_new(DTLSv1_client_method());

/*****  BOTH  *****/
/* Load certificates and key */
SSL_CTX_use_certificate_chain_file(ctx, "cert.pem");
SSL_CTX_use_PrivateKey_file(ctx, "key.pem", SSL_FILETYPE_PEM);

/* Server: Client has to authenticate */
/* Client: verify server's certificate */
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, verify_cert);

SSL_CTX_set_cookie_generate_cb(ctx, generate_cookie);
SSL_CTX_set_cookie_verify_cb(ctx, verify_cookie);
```

Note that three callback functions have been used, that is verify_cert(), generate_cookie() and verify_cookie(). The first function, verify_cert(), is called

every time a certificate has been received. This function has to verify the certificate and returns 1 if trusted or 0 otherwise. Usually the program will print certificate details and ask the user if he trusts it, or maintains a database of known certificates. In case the certificate is not trusted, the handshake and therefore the connection setup will fail. The other callback functions, generate_cookie() and verify_cookie(), are used for the cookie handling. When a cookie has to be generated for a *HelloVerifyRequest*, the generate_cookie() function is called and after receiving a cookie attached to a *ClientHello* the verify_cookie() function. The content is arbitrary, but for security reasons it should contain the client's address, a timestamp and should be signed. The signatures of the callback functions are as follows:

```
/* Certificate verification. Returns 1 if trusted, else 0 */
int verify_cert(int ok, X509_STORE_CTX *ctx);

/* Generate cookie. Returns 1 on success, 0 otherwise */
int generate_cookie(SSL *ssl, unsigned char *cookie,
                    unsigned int *cookie_len);

/* Verify cookie. Returns 1 on success, 0 otherwise */
int verify_cookie(SSL *ssl, unsigned char *cookie,
                  unsigned int cookie_len);
```

Connection setup

The server needs a socket for awaiting incoming connections. For this socket a BIO object has to be created, which can then be used with an SSL object to respond to connection attempts. To prevent DOS attacks, the server should use the *HelloVerifyRequest* to verify the client's address. Since this is unique to DTLS, there are newly added functions to realize this.

```
int fd = socket(AF_INET6, SOCK_DGRAM, 0);
bind(fd, &server_addr, sizeof(struct sockaddr_in6));

while(1) {
   BIO *bio = BIO_new_dgram(fd, BIO_NOCLOSE);

   SSL *ssl = SSL_new(ctx);
   SSL_set_bio(ssl, bio, bio);

   /* Enable cookie exchange */
   SSL_set_options(ssl, SSL_OP_COOKIE_EXCHANGE);

   /* Wait for incoming connections */
   while (!DTLSv1_listen(ssl, &client_addr));

   /* Handle client connection */
   ...
}
```

At first, BIO_new_dgram() is used instead of BIO_new() to create a UDP specific BIO. Then a new SSL object is created using the previously set up context, to which the BIO object is assigned. The cookie exchange is not enabled by default and has to be enabled with the corresponding option. The new function DTLSv1_listen() waits for incoming *ClientHellos* on the listening socket, responds with a *HelloVerifyRequest* and returns 0, which indicates that no client has been verified yet and it needs to be called again to continue listening. When the client repeats its *ClientHello* with a valid cookie attached, the function will return 1 and the sockaddr structure of the verified client. The sockaddr structure can be used to create a new socket, connected to this client, which is used to replace the listening socket in the BIO object. Hereafter the SSL object can be used for this connection, preferably in a new thread, while new BIO and SSL objects have to be created for the listening socket, to continue listening.

```
/* Handle client connection */
int client_fd = socket(AF_INET6, SOCK_DGRAM, 0);
bind(client_fd, &server_addr, sizeof(struct sockaddr_in6));
connect(client_fd, &client_addr, sizeof(struct sockaddr_in6));

/* Set new fd and set BIO to connected */
BIO *cbio = SSL_get_rbio(ssl);
BIO_set_fd(cbio, client_fd, BIO_NOCLOSE);
BIO_ctrl(cbio, BIO_CTRL_DGRAM_SET_CONNECTED, 0, &client_addr);

/* Finish handshake */
SSL_accept(ssl);
```

Since the handshake has only been performed until the repeated *ClientHello*, SSL_accept() to complete the handshake still has to be called, before sending and receiving data.

Connecting the client to a server is rather straightforward. A socket connected to the server has to be created and put into a corresponding BIO object, which itself is used by an SSL object.

```
int fd = socket(AF_INET6, SOCK_DGRAM, 0);
connect(fd, &server_addr, sizeof(struct sockaddr_in6));

BIO *bio = BIO_new_dgram(fd, BIO_NOCLOSE);
BIO_ctrl(cbio, BIO_CTRL_DGRAM_SET_CONNECTED, 0, &server_addr);

SSL *ssl = SSL_new(ctx);
SSL_set_bio(ssl, bio, bio);

/* Perform handshake */
SSL_connect(ssl);
```

Sending & Receiving

Sending and receiving with DTLS is just the same as with TLS. The functions used are SSL_write() for sending and SSL_read() for receiving. Both return the number of bytes sent and received, respectively. In case -1 is returned, an error handling is necessary, because there are several reasons why this could have happened. The function SSL_get_error() determines if and what kind of error occurred. This is the same for sending and receiving, and should be done after every SSL_read() and SSL_write() call.

Return value	Description
SSL_ERROR_NONE	No error.
SSL_ERROR_ZERO_RETURN	Transport connection closed.
SSL_ERROR_WANT_READ SSL_ERROR_WANT_WRITE	Reading/Writing had to be interrupted, just try again.
SSL_ERROR_WANT_CONNECT, SSL_ERROR_WANT_ACCEPT	Connecting/Accepting had to be interrupted, just try again.
SSL_ERROR_WANT_X509_LOOKUP	Interrupt for certificate lookup. Try again.
SSL_ERROR_SYSCALL	Socket error.
SSL_ERROR_SSL	SSL protocol error, connection failed.

The return value SSL_ERROR_SYSCALL indicates that a problem occurred while calling recvfrom() or sendto() internally. The kind of error can be determined with the *errno* variable. Usually, a socket error is fatal and the connection cannot be continued, for example after ENOMEM, that is no memory left. However, some errors, like ECONNRESET (“Connection reset by peer”), may be ignored. This error only occurs when the peer closed its port, thus dropped a packet and notifies this with an Internet Control Message Protocol (ICMP) message. Such a message can easily be faked by an attacker to shut down the connection. Instead, the *Heartbeat Extension* should be used to check the peer's availability.

Timer and Socket Timeout Handling

To set socket timeouts, the function BIO_ctrl() should be used with the corresponding BIO object:

```
struct timeval timeout;
timeout.tv_sec = 5;
timeout.tv_usec = 0;
BIO_ctrl(bio, BIO_CTRL_DGRAM_SET_RECV_TIMEOUT, 0, &timeout);
```

Whenever a socket timeout occurs, that is EAGAIN or EWOULDBLOCK is returned, the SSL_read() or SSL_write() call will return SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. So to determine if this error was really caused by a socket timeout, the BIO object has to be asked:

```
int len = SSL_read(ssl, buffer, sizeof(buffer));
switch (SSL_get_error(ssl, len)) {
   ...
   case SSL_ERROR_WANT_READ:
      /* Handle socket timeouts */
      if (BIO_ctrl(bio, BIO_CTRL_DGRAM_GET_RECV_TIMER_EXP,
                   0, NULL)) {
         num_timeouts++;
      }
      break;
   ...
}
```

Besides the handling of socket timeouts, DTLS has also handshake timers which have to be considered. When socket timeouts are set, DTLS will automatically adjust them while handshaking if they expire too late, so the blocking call will return and retransmissions can be performed. After the handshake has been done, the socket timeouts are reset to the previous values. However, this does not work with non-blocking sockets, because no DTLS function will be called if there is no incoming or outgoing traffic. So when using non-blocking calls with select(), its timeout has to be set accordingly with the function DTLSv1_get_timeout(), which will return the time until the next timer expires, if any is running. In that case, DTLSv1_handle_timeout() must be called to perform retransmissions:

```
struct timeval timeout;
DTLSv1_get_timeout(ssl, &timeout);
int num = select(FD_SETSIZE, &rsocks, NULL, NULL, &timeout) {
   /* Handle timeouts */
   if (num == 0) {
      DTLSv1_handle_timeout(ssl);
   }
   ...
}
```

For simplicity, no socket timeouts should be set before the initial handshake is done with SSL_connect() and SSL_accept(), because if the socket timeouts expire earlier than the handshake timeouts, additional error handling will be necessary to resume the handshake in that case.

SCTP specific API

To use DTLS with SCTP, a corresponding BIO object is necessary. An SCTP-aware BIO object can be created with the function BIO_dgram_sctp_new(). Since SCTP supports one-to-one style connections, DTLSv1_listen() must not be used, and the connection handling can just be done by creating a new socket for each incoming connection with accept() and calling SSL_accept() afterwards to perform the initial handshake. To make use of SCTP's additional features, that is notifications and streams, the API of the BIO object has been extended.

SCTP supports notifications, which are informational messages sent by the protocol stack via the socket read call. They cannot be passed to the DTLS layer for decryption because they are neither encrypted nor a valid DTLS message and thus would be discarded. To retrieve notifications anyway, a callback function can be registered with the BIO object, which is then called for every incoming notification.

```
void notifications(BIO *bio, void *context, void *buffer) {
   SSL *ssl = (SSL*) context;
   ...
}

void *context = (void*) ssl;
BIO_dgram_sctp_notification_cb(bio, &notifications, context);
```

The context is an arbitrary pointer which will be passed with every call. This can be used to pass the SSL object which the occurring notification belongs to, for example.

To make use of multi-streaming and other features of SCTP, the user needs to get and set additional information and the flags passed with a send/receive socket call, to read for example which stream has been used for the received message and to set the stream on which the next message should be sent. The BIO_ctrl() function provides several options for getting and setting appropriate structures:

Option	**Description**
BIO_CTRL_DGRAM_SCTP_GET_SNDINFO	Get sndinfo for next messages sent.
BIO_CTRL_DGRAM_SCTP_SET_SNDINFO	Set sndinfo for next messages sent.
BIO_CTRL_DGRAM_SCTP_GET_RCVINFO	Get rcvinfo for last message received.
BIO_CTRL_DGRAM_SCTP_SET_RCVINFO	Set rcvinfo for last message received.
BIO_CTRL_DGRAM_SCTP_GET_PRINFO	Get prinfo for next messages sent.
BIO_CTRL_DGRAM_SCTP_SET_PRINFO	Set prinfo for next messages sent.

The structures used are defined as follows:

```
/* Information used for sending */
struct bio_dgram_sctp_sndinfo
{
   uint16_t snd_sid;
   uint16_t snd_flags;
   uint32_t snd_ppid;
   uint32_t snd_context;
};
```

```
/* Information after receiving */
struct bio_dgram_sctp_rcvinfo
{
   uint16_t rcv_sid;
   uint16_t rcv_ssn;
   uint16_t rcv_flags;
   uint32_t rcv_ppid;
   uint32_t rcv_tsn;
   uint32_t rcv_cumtsn;
   uint32_t rcv_context;
};

/* Configuring PR-SCTP */
struct bio_dgram_sctp_prinfo
{
   uint16_t pr_policy;
   uint32_t pr_value;
};
```

This examples shows how to use the BIO_ctrl() call and the listed options to retrieve the additional information SCTP passes with each received message:

```
struct bio_dgram_sctp_rcvinfo rcvinfo;
BIO_ctrl(bio, BIO_CTRL_DGRAM_SCTP_GET_RCVINFO,
         sizeof(struct bio_dgram_sctp_rcvinfo), &rcvinfo);
printf("Received message on stream %d.", rvcinfo.rcv_sid);
```

Conclusion

The widely deployed TLS works without limitations only with TCP, which is not the preferred transport protocol in all scenarios. To secure an unreliable protocol like UDP or one with special features like SCTP, its modification DTLS can be used. All dependencies to the transport protocol have been removed, so DTLS does not require any feature but the transport itself.

OpenSSL contains a DTLS implementation for UDP since release 0.9.8, with major improvements since release 1.0.0. Heartbeat and SCTP support is available as a patch and is planned to be included in release 1.0.1.

With the inclusion of DTLS in OpenSSL, it is available on many different platforms and even by default on most open source operating systems. Furthermore, applications that want to make use of DTLS without relying on the operating system can already be deployed as a static build with the latest OpenSSL release. This even allows to use DTLS on restricted mobile platforms like Apple's iOS and Google's Android.

References

[1] T. Dierks; E. Rescorla. *The Transport Layer Security (TLS) Protocol*. IETF RFC 5246 - August 2008

[2] R. Stewart. *Stream Control Transmission Protocol*. IETF RFC 4960 - September 2007

[3] E. Rescorla; N. Modadugu. *Datagram Transport Layer Security*. IETF RFC 4347 - April 2006

[4] R. Seggelmann; M. Tüxen; M. Williams. *Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension*. IETF draft-ietf-tls-dtls-heartbeat-02 (work in progress) - February 2011

[5] R. Seggelmann; M. Tüxen; E.P. Rathgeb. *Design and Implementation of SCTP-aware DTLS*. Proceedings of the International Conference on Telecommunication and Multimedia (TEMU), Greece - July 2010

[6] M. Tüxen; R. Seggelmann; E. Rescorla. *Datagram Transport Layer Security (DTLS) for Stream Control Transmission Protocol (SCTP)*. IETF RFC 6083 - December 2010

[7] N. Modadugu; E. Rescorla. *The Design and Implementation of Datagram TLS*, Proceedings of the Network and Distributed System Security Symposium (NDSS), USA - February 2004

[8] http://sctp.fh-muenster.de/dtls-patches.html

[9] http://sctp.fh-muenster.de/dtls-samples.html