Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 3, 2018

O. Friel
R. Barnes
M. Pritikin
Cisco

October 30, 2017

Application-Layer TLS
draft-friel-tls-over-http-00

Abstract

Many clients need to establish secure connections to application services but face challenges establishing these connections due to the presence of middleboxes that terminate TLS connections from the client and restablish new TLS connections to the service. This document defines a mechanism for transporting TLS records in HTTP message bodies between clients and services. This enables clients and services to establish secure connections using TLS at the application layer, and treat any middleboxes that are intercepting traffic at the network layer as untrusted transport. In short, this mechanism moves the TLS handshake up the OSI stack to the application layer.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 3, 2018.

Copyright Notice

Table of Contents

1. Introduction

There are far more classes of clients being deployed on today's networks than at any time previously. This poses challenges for network administators who need to mange their network and the clients connecting to their network, and poses challenges for client vendors and client software developers who must ensure that their clients can connect to all required services.

One common example is where a client is deployed on a local domain network that protects its perimeter using a TLS terminating middlebox, and the client needs to establish a secure connection to a service in a different network via the middlebox. Traditionally, this has been enabled by the network administrator deploying the necessary certificate authority trusted roots on the client. This can be achieved at scale using standard tools that enable the administrator to automatically push trusted roots out to all client machines in the network from a centralised domain controller. This works for for personal computers, laptops and servers running standard Operating Systems that can be centrally managed. This client management process breaks for multilpe classes of clients that are being deployed today, there is no standard mechanism for configuring trusted roots on these clients, and there is no standard mechaism for these clients to securely traverse middleboxes.

The TLS over HTTP mechanism defined in this document enables clients to traverse middleboxes that restrict communications to HTTP traffic they have inserted themselves into, and establish secure connections to services across network domain boundaries.

2. Terminology

TLS over HTTP is referred to as ATLS throughout this document i.e. "Application Layer TLS".

3. ATLS Transport Goals

The high level goals driving the design of this mechanism are:

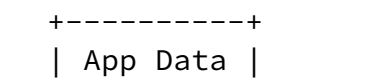
- reuse existing TLS specifications [RFC5246] [I-D.ietf-tls-tls13] as is without requiring any protocol changes
- work with all versions of TLS
- do not require any changes to current TLS software stacks
- do not mandate constraints on how the TLS stack is configured or used

- be forward compatible with future TLS versions
- work with both HTTP and HTTPS transport
- avoid introducing TLS protocol handling logic or semantics into the HTTP application layer i.e. TLS protocol knowledge and logic is handled by the TLS stack, HTTP is just a dumb transport
- ensure the client and server software implementations are as simple as possible

4. Architecture Overview

4.1. Network Architecture

A typical network deployment is illustrated in Figure 1. It shows a client connecting to a service via a middlebox. It also shows a TLS terminator deployed in front of the service. The client establishes a transport layer TLS connection with the middlebox (C->M TLS), the middlebox in turn opens a transport layer TLS connection with the TLS terminator deployed in front of the service (M->T TLS). The client can ignore any certificate validation errors when it connects to the middlebox. HTTP messages are transported over this layer between the client and the service. Finally, application layer TLS messages are exchanged inside the HTTP message bodies in order to establish an end-to-end TLS session between the client and the service (C->S TLS).



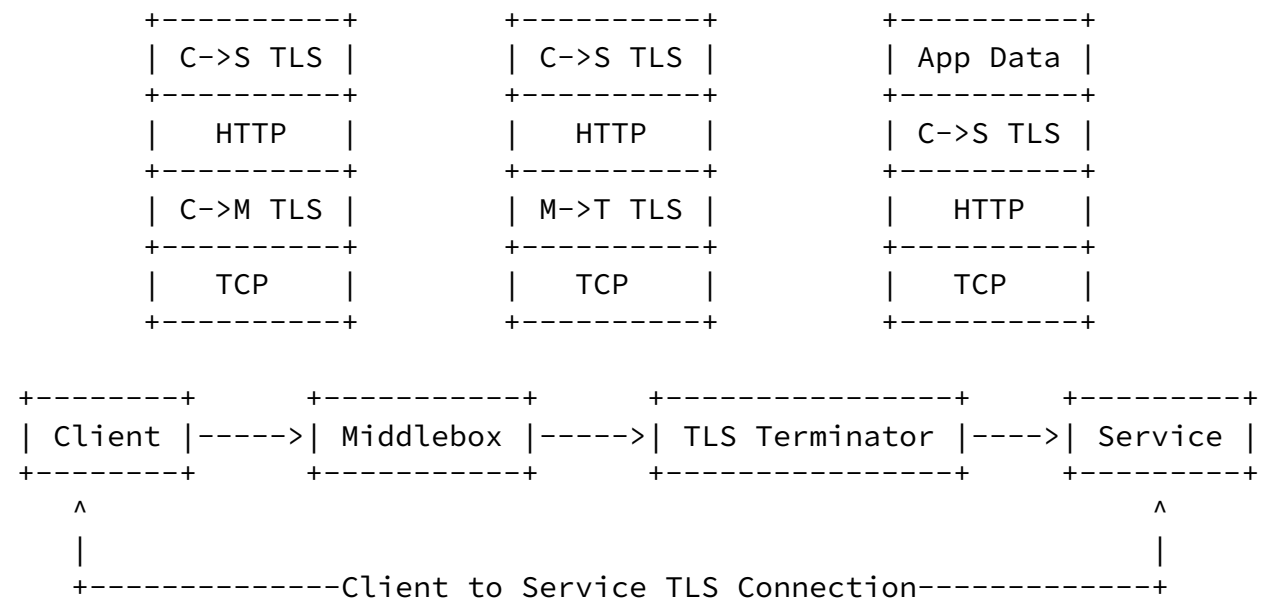


Figure 1: Network Architecture

4.2. Application Architecture

TLS software stacks allow application developers to 'unplug' the default network socket transport layer and read and write TLS records directly from byte buffers. This enables application developers to create application layer TLS sessions, extract the raw TLS record bytes from the bottom of the TLS stack, and transport these bytes over any suitable transport. The TLS software stacks can generate byte streams of full TLS flights which may include multiple TLS records. This is illustrated in Figure 2 below.

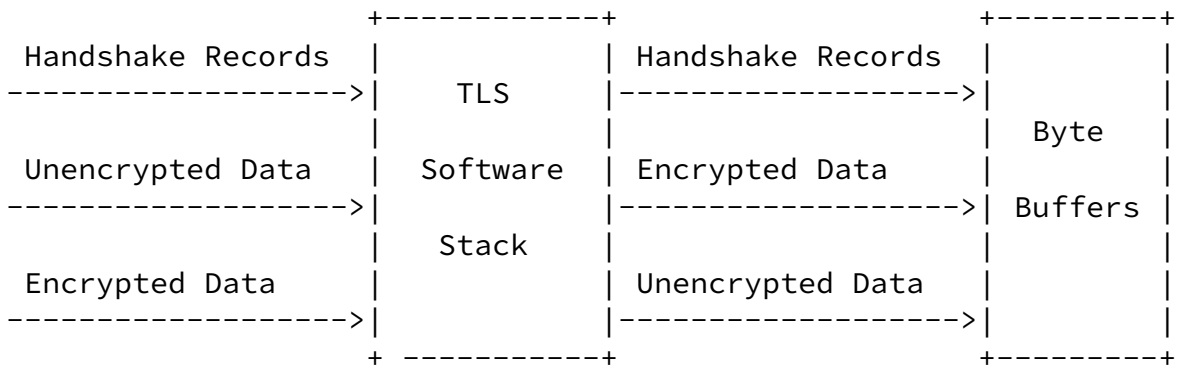


Figure 2: TLS Stack Interfaces

These TLS software stack APIs enable application developers to build the software architecture illustrated in Figure 3. The application creates and interacts with an application layer TLS session in order to generate and consume raw TLS records. The application transports these raw TLS records inside HTTP message bodies using a standard HTTP stack. The HTTP stack may in turn use either TLS or TCP transport to comunicate with the peer. The application layer TLS session and network layer TLS session can both leverage a shared, common TLS software stack. This high level architecture is applicable to both clients and services.

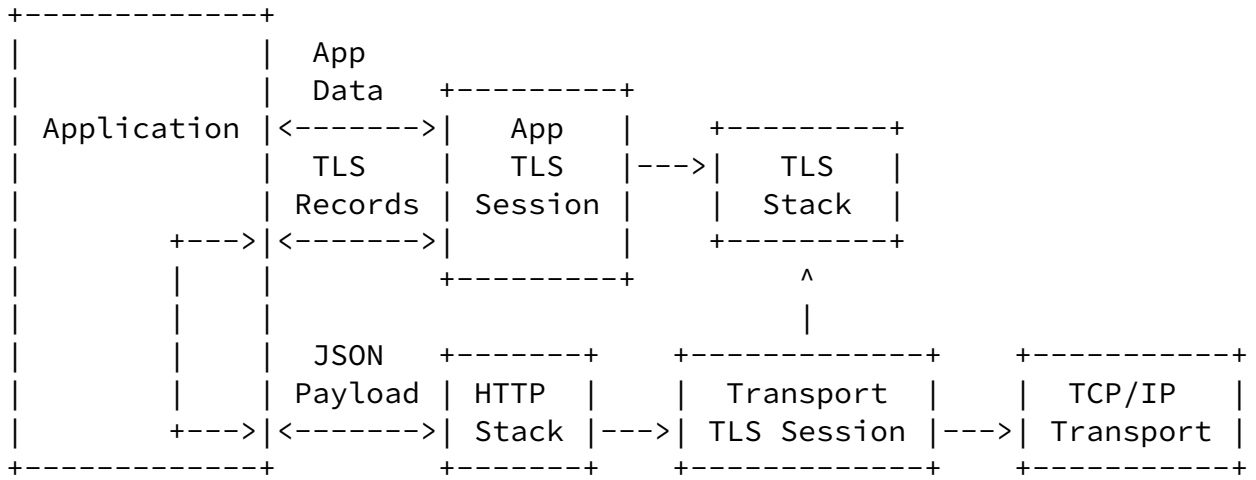


Figure 3: Application Architecture

4.3. Application Arthiecture Benefits

There are several benefits to using a standard TLS software stack to establish an application layer secure communications channel between a client and a service. These include:

- no need to define a new cryptographic negotiation and exchange protocol beween client and service
- automatically benefit from new cipher suites by simply upgrading the TLS software stack
- automaticaly benefit from new features, bugfixes, etc. in TLS software stack upgrades

4.4. Implementation

Pseudo code illustrating how to read and write TLS records directly from byte buffers using both OpenSSL and Java JSSE is given in the appendices.

5. ATLS Overview

The assumption is that the client will establish a transport layer connection to the server for exchange of HTTP messages. The underlying transport layer connection could be over TCP or TLS. The client will then establish an application layer TLS connection with the server by exchanging TLS records with the server inside HTTP message requests and responses.

5.1. TLS Connections

If the underlying transport layer connection is TLS, this means that the client will establish two independent TLS connections:

- one at the transport layer which could be directly with the service or could be with a middlebox

- one at the application layer which will be with the service

As an optimisation, clients may choose to only use ATLS as a fallback mechanism if certificate validation fails on the transport layer TLS connection to the service. For the purposes of establishing a secure connection with the service, the client does not need to perform any certificate checks or validation on the transport layer TLS connection.

Similarly, the service may also establish two independent TLS connections:

- one at the transport layer which could be directly with the client or could be with a middlebox

- one at the application layer which will be with the client

Once the application layer TLS connection is estabilshed, the client may report to the service the TLS certificates that where presented by the network layer TLS connection but this is application specific behaviour and outside the scope of this specification.

5.2. Protocol Introduction

All application TLS records are transported as base64 encoded payloads inside JSON message bodies over HTTP transport. Each payload contains a full TLS flight made up of one or more TLS records.

The client sends all application TLS records to the server in JSON message bodies in POST requests.

The server sends all TLS records to the client in JSON message bodies in 200 OK responses to the POST requests.

No constraints are placed on the ContentType contained within the transported TLS records. The TLS records may contain handshake, application_data, alert or change_cipher_spec messages. If new ContentType messages are defined in future TLS versions, these may also be transported using this protocol.

If the server is able to handle the application layer TLS records

included in the request, the server always responds with a 200 OK and includes any application TLS records in the message body. The server does not, for example, parse the TLS records generated by its TLS software stack for an AlertDescription and attempt to map this to a suitable HTTP error response code.

The server only responds with a non-200 OK message if a server error occurrs and it is not capable of handling the application layer TLS message received from the client.

5.3. TLS Session Tracking

The service needs to track multiple client application layer TLS sessions so that it can collerate TLS records received in HTTP message bodies with the appropriate TLS session. It does this by inclusion of an explicit session identifier in the JSON message body.

5.4. Upgrade to Websocket

The HTTP connection between the client and the service may be upgraded to a websocket if required. This would allow a server to send a TLS close request, or any application data, asynchronously to the client. Note that for the majority of use cases, there will be no need to open a websocket between the client and service.

5.5. Service Container Affinity

Application services are typically distributed across multiple containers and virtual machines. As TLS is stateful, it must be ensured that sequences of TLS messages are handled appropriately by the service deployment and the service execution engine has access to all necessary state information. This is explicilty outside the scope of this specification as there are multiple well defined mechanisms for enabling this.

5.6. Keying Material Exporting

This specification does not require, or preclude, the use of [RFC5705]. When the client and service applications detect that the ATLS session is established, the application may use the key exporter functions of the TLS stack to derive shared keys between client and service. The client and service may then use these shared keys to establish an independent cryptographic context and exchange data using any suitable mechanism such as JSON Web Encryption [RFC7516] or Encrypted Content-Encoding for HTTP [RFC8188].

6. Protocol Details

6.1. Message Body

All message bodies are JSON bodies containing one or two parameters:

```
{
  "session": "<session-string>",
  "records": "<base64 encoded TLS records>"
}
```

The following two parameters are defined:

session: This is set by the service and is used to correlate requests across multiple client sessions. This parameter is included in all messages apart from the first first message sent from the client to the service. When a client sends the first request to establish an ATLS session with a service, it MUST omit this parameter. When a service handles the first request from a client (and that request will include the ClientHello), and the service creates an internal TLS session object, it MUST return a server-generated "<session-string>" to the client. The client MUST include that "<session-string>" in all subsequent messages to the server. If the service is unable to find a TLS session that correlates with the "<session-string>" that a client specifies, the server MUST return 422 Unprocessable Entity.

records: This parameter is used to transport the base64 encoded TLS records that the client and service applications retrieve from their TLS stack. This parameter is sent in all requests from the client to the service. This parameter may not necessarily be sent in all response messages from the service to the client if the service has no TLS records to send. This can happen with a TLS1.3 handshake.

6.2. HTTP Content-Type

A new HTTP Content-Type is defined:

Content-Type: application/atls+json

6.3. Client Requests

When a client has base64 encoded TLS records to send to a service, it will include the previously received "<session-string>" in the request, or else omit this field for the very first handshake message, and send the following request to the service:

```
POST /atls
Content-Type: application/atls+json

{
  "session": "<session-string>",
  "records": "<base64 encoded TLS records>"
}
```

6.4. Server Responses

When a service has processed the TLS records received from a client and has generated TLS records to reply with, it will send the following reply to the client:

```
200 OK
Content-Type: application/atls+json

{
  "session": "<session-string>",
  "records": "<base64 encoded TLS records>"
}
```

The server MUST respond with one of the following status codes:

200 OK: The server was able to successfully parse the request and process the TLS records using its TLS software stack.

400 Bad Request: The client's request did not contain a JSON object of the form specified above.

422 Unprocessable Entity: The client presented a "<session-string>" that the service is unable to correlate that to an existing TLS session.

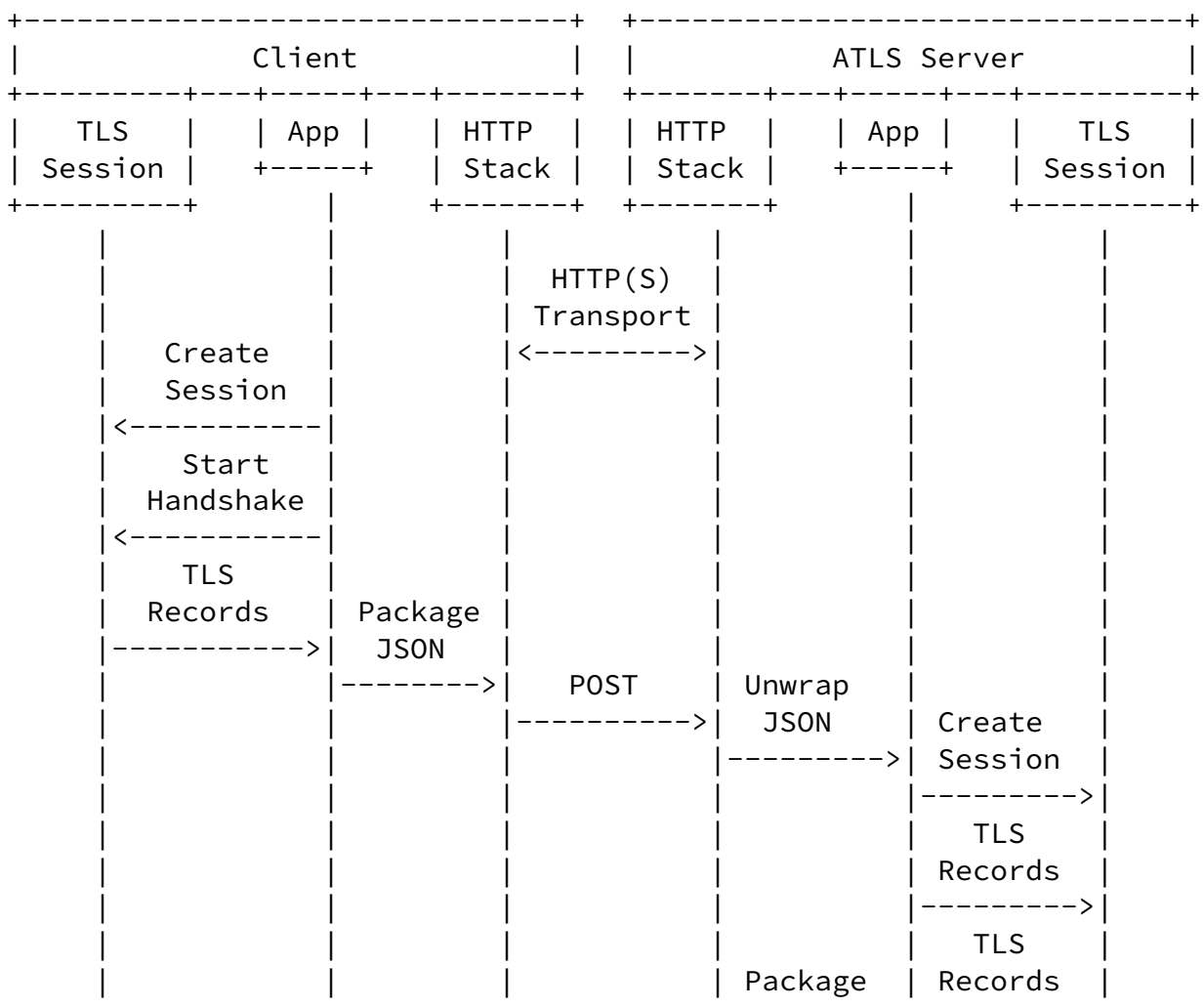
Note that a status code of 200 OK does not indicate that the TLS connection being negotiated is error-free. Alerts produced by TLS will be returned in the encoded TLS records. A 200 OK response simply indicates that the client should provide the records encoded in the response to its TLS stack.

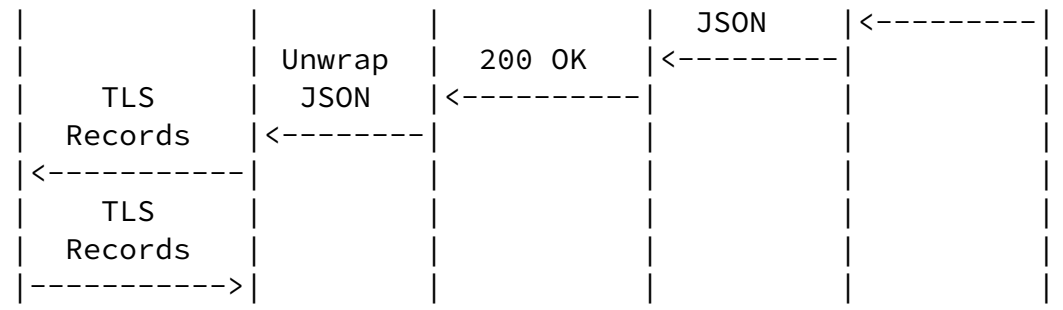
7. ATLS Session Establishment

This section describes a typical ATLS session establishment flow.

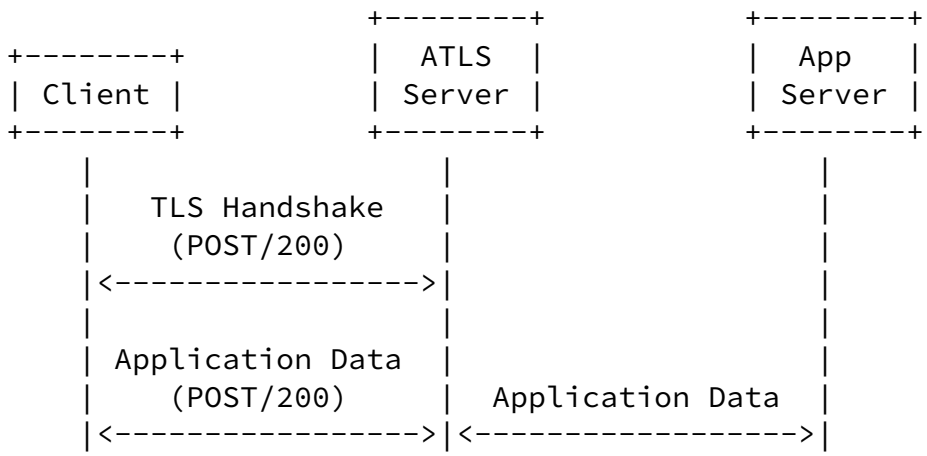
7.1. ATLS Handshake Message Sequence Flow

The following flow chart shows an illustrative message sequence flow for the first TLS handshake flight between a client and a service.





This process repeats until the handshake completes. Once the application-layer TLS connection is ready to carry application data, the ATLS server relays it to the application server that is ultimately serving the rqeuest. That is, a given ATLS server is assumed to be connected to a single application endpoint.



7.2. Detailed ATLS Handshake

o Client establishes transport layer connection with service

This transport layer session can be established over TCP or TLS. If over TLS, the client does not need to perform certificate validation on the TLS connection, and may ignore any certificate validation errors if it does perform certificate validation.

o Client creates an application TLS session object

The client application creates a TLS session object that is not bound

to any network socket. The client initiates a TLS handshake on the session. This will result in the TLS session generating the TLS record bytes for a full TLS handshake first flight. This will be a ClientHello message. Note that the client application does not explicitly know what the contents of the TLS record bytes are.

- The client base64 encodes the TLS flight 1 records and sends them in a HTTP POST to the service

```
POST /atls
Content-Type: application/atls

{
  "records": "<base64 encoded flight 1>"
}
```

- The service creates a TLS session for handling the request

The service notes that there is no "session" in the request and creates an application TLS session object and a suitable "<session-string>" for correlation. The service decodes the received base64 encoded "records" and passes them to the TLS session. The session handles the TLS handshake message and generates a full TLS handshake

response flight. Typically, this will be a ServerHello and additional handshake messages that are TLS version dependent. For TLS1.3, this may include a Finished message. Note that the service application does not explicitly know what the contents of the TLS record bytes are.

- The service base64 encodes the TLS flight 1 response records and sends them in the response to the client

The service MUST include its generated "session".

```
200 OK
Content-Type: application/atls

{
  "session": "<session-string>",
  "records": "<base64 encoded flight 1 response>"
}
```

- The client passes the service response to its TLS session

The client decodes the received base64 encoded "records" and passes them to the TLS session. The session handles the TLS handshake message and generates a full TLS handshake second flight. This will be a Finished message, Certificate and CertificateVerify messages if required, and additional handshake messages that are TLS version dependent.

- The client base64 encodes the TLS flight 2 records and sends them in a HTTP POST to the service

The client MUST include the "<session-string>" it received from the service.

```
POST /atls
Content-Type: application/atls

{
  "session": "<session-string>",
  "records": "<base64 encoded flight 2>"
}
```

- The service passes the client response to the respective TLS session

The service extracts the "<session-string>" from the request and finds the respective application TLS session object. The service decodes the received base64 encoded "records" and passes them to the

TLS session. The session handles the TLS handshake message and will generates a full TLS handshake response flight where appropriate. For TLS1.2, this will include a Finished and ChangeCipherSpec message. For TLS1.3, there are not TLS records generated.

- The service base64 encodes the TLS flight 2 response records and sends them in the response to the client

For TLS 1.2:

```
200 OK
```

```
Content-Type: application/atls

{
  "session": "<session-string>",
  "records": "<base64 encoded flight 2 response>"
}
```

For TLS1.3:

```
200 OK
Content-Type: application/atls

{
  "session": "<session-string>"
}
```

o The client passes the service response to its TLS session

If there are TLS records included in the response from the service, the client decodes the received base64 encoded "records" and passes them to its TLS session.

7.3. Application Data Exchange

Application data is exchanged between the client and service inside the TLS tunnel using exactly the same JSON transport payload. When the client has data to send to the service, it encrypts the data using the standard TLS stack methods (e.g. OpenSSL SSL_write() or Java SSLEngine.wrap()), extracts the encrypted TLS records from the bottom of the TLS stack, and sends them as in the JSON "records" parameter to the service. The service injects the TLS records into its stack and reads the decrypted data from the top of its stack.

8. RTT Considerations

The number of RTTs that take place when establishing a TLS session depends on the version of TLS and what capabilities are enabled on

the TLS software stack. For example, a 0-RTT exchange is possible with TLS1.3.

If applications wish to ensure a predictable number of RTTs when establishing an application layer TLS connection, this may be achieved by configuring the TLS softwrae stack appropriately. Relevant configuration parameters for OpenSSL and Java SunJSSE stacks are outlined in the appendix.

9. IANA Considerations

[[TODO - New Content-Type must be registered.]]

10. Security Considerations

[[TODO]]

11. Appendix A. TLS Software Stack Configuration

[[EDITOR'S NOTE: We could include details here on how TLS stack configuration items control the number of round trips between the client and server.
And just give two examples: OpenSSL and Java SunJSSE]]

12. Appendix B. Pseudo Code

This appendix gives both C and Java pseudo code illustrating how to inject and extract raw TLS records from a TLS software stack. Please not that this is illustrative, non-functional pseudo code that does not compile. Functioning proof-of-concept code is available on the following public respository [[EDITOR'S NOTE: Add the URL here]].

12.1. B.1 OpenSSL

OpenSSL provides a set of Basic Input/Output (BIO) APIs that can be used to build a custom transport layer for TLS connections. This appendix gives pseudo code on how BIO APIs could be used to build a client application that completes a TLS handshake and exchanges application data with a service.

```
   char inbound[MAX];
   char outbound[MAX];
   int rx_bytes;
   SSL_CTX *ctx = SSL_CTX_new();
   SSL *ssl = SSL_new(ctx);

   // Create in-memory BIOs and plug in to the SSL session
   BOI* bio_in = BIO_new(BIO_s_mem());
   BOI* bio_out = BIO_new(BIO_s_mem());
   SSL_set_bio(ssl, bio_in, bio_out);

   // We are a client
   SSL_set_connect_state(ssl);

   // Loop through TLS flights until we are done
   do {
     // Calling SSL_do_handshake() will result in a full
     // TLS flight being written to the BIO buffer
     SSL_do_handshake(ssl);

     // Read the client flight that the TLS session
     // has written to memory
     BIO_read(bio_out, outbound, MAX);

     // POST the outbound bytes to the server using a suitable
     // function. Lets assume that the server response will be
     // written to the 'inbound' buffer
     num_bytes = postTlsRecords(outbound, inbound);

     // Write the server flight to the memory BIO so the TLS session
     // can read it. The next call to SSL_do_handshake() will handle
     // this received server flight
     BIO_write(bio_in, inbound, num_bytes);

   } while (!SSL_is_init_finished(ssl));

   // Send a message to the server. Calling SSL_write() will run the
   // plaintext through the TLS session and write the encrypted TLS
   // records to the BIO buffer
   SSL_write(ssl, "Hello World", strlen("Hello World"));

   // Read the TLS records from the BIO buffer and
   // POST them to the server
   BIO_read(bio_out, outbound, MAX);
   num_bytes = postTlsRecords(outbound, inbound);
```

12.2. B.2 Java JSSE

The Java SSLEngine class "enables secure communications using protocols such as the Secure Sockets Layer (SSL) or IETF RFC 2246 "Transport Layer Security" (TLS) protocols, but is transport independent". This pseudo code illustrates how a server could use the SSLEngine class to handle an inbound client TLS flight and generate an outbound server TLS flight response.

```
SSLEngine sslEngine = SSLContext.getDefault().createSSLEngine();
sslEngine.setUseClientMode(false);
sslEngine.beginHandshake();

// Lets assume 'inbound' has been populated with
// the Client 1st Flight
ByteBuffer inbound;

// 'outbound' will be populated with the
// Server 1st Flight response
ByteBuffer outbound;

// SSLEngine handles one TLS Record per call to unwrap().
// Loop until the engine is finished unwrapping.
while (sslEngine.getHandshakeStatus() ==
       HandshakeStatus.NEED_UNWRAP) {
  SSLEngineResult res = sslEngine.unwrap(inbound, outbound);

  // SSLEngine may need additional tasks run
  if (res.getHandshakeStatus() == NEED_TASK) {
    Runnable run = sslEngine.getDelegatedTask();
    run.run();
  }
}

// The SSLEngine has now finished handling all inbound TLS Records.
// Check if it wants to generate outbound TLS Records. SSLEngine
// generates one TLS Record per call to wrap().
// Loop until the engine is finished wrapping.
while (sslEngine.getHandshakeStatus() ==
       HandshakeStatus.NEED_WRAP) {
  SSLEngineResult res = sslEngine.wrap(inbound, outbound);
```

```
    // SSLEngine may need additional tasks run
    if (res.getHandshakeStatus() == NEED_TASK) {
      Runnable run = sslEngine.getDelegatedTask();
      run.run();
    }
  }

  // outbound ByteBuffer now contains a complete server flight
  // containing multiple TLS Records
  // Rinse and repeat!
```


13. Appendix C. Example ATLS Handshake

[[EDITOR'S NOTE: For completeness, include a simple full TLS handshake showing the B64 encoded flights in JSON, along with the HTTP request/response/headers. And also the raw hex TLS records showing protocol bits]]

14. Informative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-21 (work in progress), July 2017.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[RFC5705] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, DOI 10.17487/RFC5705, March 2010, <https://www.rfc-editor.org/info/rfc5705>.

[RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <https://www.rfc-editor.org/info/rfc7516>.

[RFC8188] Thomson, M., "Encrypted Content-Encoding for HTTP", RFC 8188, DOI 10.17487/RFC8188, June 2017, <https://www.rfc-editor.org/info/rfc8188>.

Authors' Addresses

Owen Friel
Cisco

Email: ofriel@cisco.com

Richard Barnes
Cisco

Email: rlb@ipv.sx

Max Pritikin
Cisco

Email: pritikin@cisco.com