Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: August 4, 2018

O. Friel
R. Barnes
M. Pritikin
Cisco
H. Tschofenig
ARM Limited
M. Baugher
Consultant

January 31, 2018

Application-Layer TLS (ATLS)
draft-friel-tls-atls-00

Abstract

This document specifies how TLS sessions can be established at the application layer over untrusted transport between clients and services for the purposes of establishing secure end-to-end encrypted communications channels. Transport layer encodings for application layer TLS records are specified for HTTP and CoAP transport. Explicit identification of application layer TLS packets enables middleboxes to provide transport services and enforce suitable transport policies for these payloads, without requiring access to the unencrypted payload content. Multiple scenarios are presented identifying the need for end-to-end application layer encryption between clients and services, and the benefits of reusing the well-defined TLS protocol, and a standard TLS stack, to accomplish this are described. Application software architectures for building, and network architectures for deploying application layer TLS are outlined.

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 http://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 August 4, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction

There are multiple scenarios where there is a need for application layer end-to-end security between clients and application services. Two examples include:

o Bootstrapping devices that must connect to HTTP application services across untrusted TLS interception middleboxes

o Constrained devices connecting via gateways to application services, where different transport layer protocols may be in use on either side of the gateway, with the gateway transcoding between the different transport layer protocols.

These two scenarios are described in more detail in Section 3.

Related to this document, there is ongoing work across the industry to define requirements for end-to-end security. [I-D.hartke-core-e2e-security-reqs] documents requirements for CoAP [RFC7252] End-to-End Security. The Open Mobile Alliance (OMA) has published a candidate standard Lightweight Machine to Machine Requirements [LwM2M] which defines multiple requirements for end-to-end security.

This document describes how clients and applications can leverage standard TLS software stacks to establish secure end-to-end encrypted connections at the application layer. The connections may establish TLS [RFC5246] [I-D.ietf-tls-tls13] or DTLS [RFC6347] [I-D.ietf-tls-dtls13] sessions. There are multiple advantages to reuse of existing TLS software stacks for establishment of application layer secure connections. These include:

- many clients and application services already include a TLS software stack, so there is no need to include yet another software stack in the software build

- no need to define a new cryptographic negotiation, authentication, and key exchange protocol between clients and services

- provides standards based PKI mutual authentication between clients and services

- no need to train software developers on how to use a new cryptographic protocols or libraries

- automatically benefit from new cipher suites by simply upgrading the TLS software stack

- automatically benefit from new features, bugfixes, etc. in TLS software stack upgrades

This document also explicitly defines how application layer TLS connections can be established using HTTP [RFC7230] [RFC7540] or CoAP as transport layers. This document does not preclude the user of other transport layers, however defining how application layer TLS connections can be established over other transport layers such as [ZigBee] or [Bluetooth] is beyond the scope of this document.

Explicitly identifying application layer TLS packets enables transport layer middleboxes to provide transport capabilities and enforce suitable transport policies for these payloads, without requiring access to unencrypted application data.

2. Terminology

Application layer TLS is referred to as ATLS throughout this document.

3. Application Layer End-to-End Security Use Cases

This section describes in more detail the bootstrapping and constrained device use cases mentioned in the introduction.

3.1. Bootstrapping Devices

There are far more classes of clients being deployed on today's networks than at any time previously. This poses challenges for network administrators 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 TCP/IP 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. This is illustrated in Figure 1.

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 centralized 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 multiple 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 mechanism for these clients to securely traverse middleboxes.

```
+--------+    C->M TLS    +-----------+   M->S TLS   +---------+
| Client |--------------->| Middlebox |------------->| Service |
+--------+                +-----------+              +---------+
    ^                                                     ^
    |                                                     |
    +-----------Client to Service ATLS Connection---------+
```

Figure 1: Bootstrapping Devices

The ATLS mechanism defined in this document enables clients to traverse middleboxes and establish secure connections to services across network domain boundaries. The purpose of this connection may simply be to facilitate a bootstrapping process, for example [I-D.ietf-anima-bootstrapping-keyinfra], whereby the client securely discovers the local domain certificate authorities required to establish a trusted network layer TLS connection to the middlebox.

3.2. Constrained Devices

Two constrained device use cases are outlined here.

3.2.1. Constrained Device Connecting over a Closed Network

There are industry examples of home smart lighting systems where the smart light bulbs connect using ZigBee to a gateway device. A controller application running on a mobile device connects to the

gateway using CoAP over DTLS. The controller can then control the light bulbs by sending messages and commands via the gateway. The gateway device has full access to all messages sent between the light bulbs and the controller application.

A generic use case similar to the smart lighting system outlined above has an IoT device talking ZigBee to a gateway, with the gateway in turn talking CoAP over DTLS to a controller application running on a mobile device. This is illustrated in Figure 2.

There are scenarios where the messages sent between the IoT device and the controller application must not be exposed to the gateway function. Additionally, the end devices (the IoT device and the controller application service) have no visibility to and no guarantees about what transport layer security and encryption is enforced across all hops end-to-end as they only have visibility to their immediate next hop. ATLS addresses these concerns.

```
+--------+    ZigBee     +---------+  CoAP/DTLS   +------------+
| Device |-------------->| Gateway |------------->| Mobile App |
+--------+               +---------+              +------------+
    ^                                                   ^
    |                                                   |
    +--------Device to Mobile App ATLS Connection-------+
```

Figure 2: IoT Closed Network Gateway

3.2.2. Constrained Device Connecting over the Internet

A somewhat similar example has an IoT device connecting to a gateway using a suitable transport mechanism such as ZigBee, CoAP, MQTT, etc. The gateway function in turn talks HTTP over TLS (or, for example, HTTP over QUIC) to an application service over the Internet. This is illustrated in Figure 3.

The gateway may not be trusted and all messages between the IoT device and the application service must be end-to-end encrypted. Similar to the previous use case, the endpoints have no guarantees about what level of transport layer security is enforced across all hops. Again, ATLS addresses these concerns.

```
+--------+  CoAP/DTLS    +------------------+  HTTP/TLS   +---------+
| Device |-------------->| Internet Gateway |------------>| Service |
+--------+               +------------------+             +---------+
    ^                                                          ^
    |                                                          |
    +---------Device to Cloud Service ATLS Connection----------+
```

Figure 3: IoT Internet Gateway

4. Current Approaches to Application Layer End-to-End Security

End-to-end security at the application layer is increasing seen as a key requirement across multiple applications and services. Some examples of end-to-end security mechanisms are outlined here. All the solutions outlined here have some common characteristics. The solutions:

- do not rely on transport layer security

- define a new handshake protocol for establishment of a secure end-to-end session

4.1. Noise

[Noise] is a framework for cryptographic protocols based on Elliptic Curve Diffie-Hellman (ECDH) key agreement, AEAD encryption, and BLAKE2 and SHA2 hash functions. Noise is currently used by WhatsApp, WireGuard, and Lightning.

The current Noise protocol framework defines mechanisms for proving possession of a private key, but does not define authentication mechanisms. Section 14 "Security Considerations" of Noise states: ~~~ it's up to the application to determine whether the remote party's static public key is acceptable ~~~

4.2. Signal

The [Signal] protocol provides end-to-end encryption and uses EdDSA signatures, Triple Diffie-Hellman handshake for shared secret establishment, and the Double Ratchet Algorithm for key management. It is used by Open Whisper Systems, WhatsApp and Google.

Similar to Noise, Signal does not define an authentication mechanism. The current [X3DH] specification states in section 4.1 "Authentication":

Methods for doing this are outside the scope of this document

4.3. Google ALTS

Google's Application Layer Transport Security [ALTS] is a mutual authentication and transport encryption system used for securing Remote Procedure Call (RPC) communications within Google's infrastructure. ALTS uses an ECDH handshake protocol and a record protocol containing AES encrypted payloads.

4.4. Ephemeral Diffie-Hellman Over COSE

There is ongoing work to standardise [I-D.selander-ace-cose-ecdhe]. This defines a ECDH SIGMA based authenticated key exchange algorithm using COSE and COBR objects.

5. ATLS Goals

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

- enable authenticated key exchange at the application layer by reusing existing technologies

- ensure that ATLS packets are explicitly identified thus ensuring that any middleboxes or gateways at the transport layer are content aware

- leverage existing TLS stacks and handshake protocols thus avoiding introducing new software or protocol dependencies in clients and applications

- reuse existing TLS [RFC5246] [I-D.ietf-tls-tls13] and DTLS [RFC6347] [I-D.ietf-tls-dtls13] specifications as is without requiring any protocol changes or software stack changes

- do not mandate constraints on how the TLS stack is configured or used

- be forward compatible with future TLS versions

- avoid introducing TLS protocol handling logic or semantics into the application layer i.e. TLS protocol knowledge and logic is handled by the TLS stack, not the application

- ensure the client and server software implementations are as simple as possible

6. Architecture Overview

6.1. 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. Additionally, TLS software stacks support Keying Material Exporters [RFC5705] and allow applications to export keying material from established TLS sessions. This keying material can then be used by the application for encryption of data outside the context of the TLS session. This is illustrated in Figure 4 below.

```
                       +------------+                    +---------+
    Handshake Records  |            | Handshake Records  |         |
   ------------------->|            |------------------->|         |
                       |            |                    |  Byte   |
    Unencrypted Data   |    TLS     | Encrypted Data     |         |
   ------------------->|            |------------------->| Buffers |
                       |  Software  |                    |         |
    Encrypted Data     |            | Unencrypted Data   |         |
   ------------------->|   Stack    |------------------->|         |
                       |            |                    +---------+
    Keying Material    |            |
   <-------------------|            |
                       + -----------+
```

Figure 4: TLS Stack Interfaces

These TLS software stack APIs enable application developers to build the software architectures illustrated in Figure 5 and Figure 6.

In both architectures, 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 transport layer message bodies using whatever standard transport layer stack is suitable for the application or architecture. This document does not place any restrictions on the choice of transport layer and any suitable protocol such as HTTP, TCP, CoAP, ZigBee, Bluetooth, etc. could be used.

The transport layer will typically encrypt data, and this encryption is completely independent from any application layer encryption. The transport stack may create a transport layer TLS session. The

application layer TLS session and transport layer TLS session can both leverage a shared, common TLS software stack. This high level architecture is applicable to both clients and application services. The key differences between the architectures are as follows.

In the model illustrated in Figure 5, the application sends all sensitive data that needs to be securely exchanged with the peer application through the Application TLS session in order to be encrypted and decrypted. All sensitive application data is thus encoded within TLS records by the TLS stack, and these TLS records are transmitted over the transport layer.

```
+-------------+
|             |    App
|             |    Data    +---------+
| Application |<---------->|   App   |      +---------+
|             |    TLS     |   TLS   |----->|   TLS   |
|             |  Records   | Session |      |  Stack  |
|        +--->|<---------->|         |      +---------+
|        |    |            +---------+           ^
|        |    |                                  |?
|        |    | Transport +-----------+    +------------+
|        |    |  Payload  | Transport |    | Transport  |
|        +--->|<--------->|   Stack   |--->| Encryption |-->Packets
+-------------+           +-----------+    +------------+
```

Figure 5: TLS Stack used for all data encryption

In the model illustrated in Figure 6, the application establishes an application layer TLS session purely for the purposes of key exchange. Therefore, the only TLS records that are sent or received by the application layer are TLS handshake records. Once the application layer TLS session is established, the application uses Keying Material Exporter [RFC5705] APIs to export keying material from the TLS stack from this application layer TLS session. The application can then use these exported keys to derive suitable shared encryption keys with its peer for exchange of encrypted data. The application encrypts and decrypts sensitive data using these shared encryption keys using any suitable cryptographic library (which may be part of the same library that provides the TLS stack), and transports the encrypted data directly over the transport layer.

```
   +--------------+
   |              |
   | Application  |
   |              |
   | +-------+    |            +---------+
   | | App   |    | Key Export |         |
   | | Data  |<---|<-----------|         |
   | | Crypto|    |            |   App   |
   | +-------+    |    TLS     |   TLS   |      +---------+
   |    ^         | Handshake  | Session |----->|   TLS   |
   |    |         |  Records   |         |      |  Stack  |
   |    |    +--->|<---------->|         |      +---------+
   |    |    |    |            +---------+           ^
   |    |    |    |                                  |?
   |    |    |    | Transport +-----------+    +------------+
   |    |    |    |  Payload  | Transport |    | Transport  |
   |    +----+--->|<--------->|   Stack   |--->| Encryption |-->Packets
   +--------------+           +-----------+    +------------+
```

Figure 6: TLS stack used for key agreement and exporting

The choice of which application architecture to use will depend on the overall solution architecture, and the underlying transport layer or layers in use. While the choice of application architecture is outside the scope of this document, some considerations are outlined here.

- for constrained devices, every single byte of payload is important. [I-D.mattsson-core-security-overhead] analyses the overhead of TLS headers compared with OSCORE [I-D.ietf-core-object-security] illustrating the additional overhead associated with TLS headers. It may be more appropriate to use the architecture defined in Figure 6 in order to establish shared encryption keys, and then transport encrypted data directly without the overhead of unwanted TLS record headers.

- when using HTTP as a transport layer, it may be more appropriate to use the architecture defined in Figure 6 in order to avoid any TLS session vs. HTTP session affinity issues.

6.1.1. Application Architecture 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 between client and service

- automatically benefit from new cipher suites by simply upgrading the TLS software stack

- automatically benefit from new features, bugfixes, etc. in TLS software stack upgrades

6.1.2. ATLS Packet Identification

It is recommended that ATLS packets are explicitly identified by a standardized, transport-specific identifier enabling any gateways and middleboxes to identify ATLS packets. Middleboxes have to contend with a vast number of applications and network operators have difficulty configuring middleboxes to distinguish unencrypted but not explicitly identified application data from end-to-end encrypted data. This specification aims to assist network operators by explicitly identifyng ATLS packets. The HTTP and CoAP encodings documented in Section 9 and Section 10 explicitly identify ATLS packets.

6.1.3. ATLS Session Tracking

The ATLS application service establishes multiple ATLS sessions with multiple clients. As TLS sessions are stateful, the application service must be able to correlate ATLS records from different clients across the relevant ATLS sessions. The details of how session tracking is implemented are outside the scope of this document. Recommendations are given in Section 9 and Section 10, but session tracking is application and implementation specific.

6.1.4. ATLS Record Inspection

It should not be necessary for the application layer to have to inspect, parse or understand the contents of ATLS records. 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.

6.1.5. Implementation

Pseudo code illustrating how to read and write TLS records directly from byte buffers using both OpenSSL BIO functions and Java JSSE SSLEngine is given in the appendices. A blog post by [Norrell] outlines a similar approach to leveraging OpenSSL BIO functions, and Oracle publish example code for leveraging [SSLEngine].

6.2. Functional Design

[todo: insert Hannes functional design section here including the policy layers]

Policy examples:

Mention that the app layer policy could be to not do ATLS if the transport layer establishes an e2e session with the peer. e.g. for HTTP use cases where there is no middlebox and cert validation passes.

Mention that the client could report in the ATLS session any middlebox cert seen at the transport layer.

6.3. Network Architecture

An example network deployment is illustrated in Figure 7. It shows a constrained client connecting to an application service via an internet gateway. The client uses CoAP over DTLS to communicate with the gateway. The gateway extracts the messages the client sent over CoAP and sends these messages inside HTTP message bodies to the application service. It also shows a TLS terminator deployed in front of the application service. The client establishes a transport layer CoAP/DTLS connection with the gateway (C->G DTLS), the gateway in turn opens a transport layer TLS connection with the TLS terminator deployed in front of the service (G->T TLS). The client can ignore any certificate validation errors when it connects to the gateway. CoAP messages are transported between the client and the gateway, and HTTP messages are transported between the client and the service. Finally, application layer TLS messages are exchanged inside the CoAP and HTTP message bodies in order to establish an end-to-end TLS session between the client and the service (C->S TLS).

```
          +----------+        +----------+
          | App Data |        | App Data |
          +----------+        +----------+         +----------+
          | C->S TLS |        | C->S TLS |         | App Data |
          +----------+        +----------+         +----------+
          |   CoAP   |        |   HTTP   |         | C->S TLS |
          +----------+        +----------+         +----------+
          | C->G DTLS|        | M->T TLS |         |   HTTP   |
          +----------+        +----------+         +----------+
          |   UDP    |        |   TCP    |         |   TCP    |
          +----------+        +----------+         +----------+

   +--------+      +-----------+      +----------------+     +---------+
   | Client |----->|  Gateway  |----->| TLS Terminator |---->| Service |
   +--------+      +-----------+      +----------------+     +---------+
      ^                                                           ^
      |                                                           |
      +-------------Client to Service ATLS Connection-------------+
```

Figure 7: Constrained Device Gateway Network Architecture

Another typical network deployment is illustrated in Figure 8. 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).

```
          +----------+        +----------+
          | App Data |        | App Data |
          +----------+        +----------+         +----------+
          | C->S TLS |        | C->S TLS |         | App Data |
          +----------+        +----------+         +----------+
          |   HTTP   |        |   HTTP   |         | C->S TLS |
          +----------+        +----------+         +----------+
          | C->M TLS |        | M->T TLS |         |   HTTP   |
          +----------+        +----------+         +----------+
          |   TCP    |        |   TCP    |         |   TCP    |
          +----------+        +----------+         +----------+

   +--------+      +-----------+      +----------------+     +---------+
   | Client |----->| Middlebox |----->| TLS Terminator |---->| Service |
   +--------+      +-----------+      +----------------+     +---------+
      ^                                                          ^
      |                                                          |
      +-------------Client to Service ATLS Connection-------------+
```

Figure 8: HTTP Middlebox Network Architecture

7. Key Exporting and Application Data Encryption

When solutions implement the architecture described in Figure 6, they leverage [RFC5705] for key exporting from the ATLS session. The client and service then use the exported keys to derive shared encryption keys. The encryption keys are then used with a suitable cipher suite to encrypt application data for exchange with the peer.

7.1. Key Exporter Label

A new TLS Exporter Label is defined for ATLS key exporting. Its value is:

TLS Exporter Label: application-layer-tls

7.2. Cipher Suite Selection

Application layer encryption performed outside the context of the ATLS session using exported keys should use the cipher suite negotiated during ATLS session establishment.

7.3. Key Derivation

[RFC5705] key exporting functions allow specification of the number of bytes of keying material that should be exported from the TLS session. The application should export the exact number of bytes

required to generate the necessary client and server cipher suite encryption key and IV values.

Maybe need to reference the relevant sections from https://tools.ietf.org/html/draft-ietf-tls-tls13-23#section-7 and https://tools.ietf.org/html/rfc5246#section-6.3.

8. ATLS Session Establishment

Figure 9 illustrates how an ATLS session is established using the key exporting architectural model shown in Figure 6. The outline is as follows:

- the client creates an ATLS session object
- the client initiates a TLS handshake on the session
- the client extracts the TLS records for the first TLS flight (the first RTT)
- the client sends the TLS records over the transport layer to the server
- on receipt of the TLS flight, the server creates an ATLS session object
- the server injects the received TLS flight into the session
- the server extracts the TLS records for the first TLS flight response
- the server sends the TLS response records over the transport layer to the client
- the client injects the received TLS records into its TLS session completing the first full RTT
- the client and server repeat the above process and complete the second RTT
- once the ATLS session is up, both sides export keying material
- both sides now can exchange data encrypted using shared keys derived from the keying material

```
   +-------------------------------+  +-------------------------------+
   |             Client            |  |          ATLS Server          |
   +---------+---+-----+---+-------+  +-------+---+-----+---+---------+
```

```
   |  ATLS   |  | App |  |Transport|  |Transport|  | App |  |  ATLS   |
   | Session |  +-----+  |  Stack  |  |  Stack  |  +-----+  | Session |
   +---------+     |     +---------+  +---------+     |     +---------+
        |          |         |             |          |          |
        |          |         |             |          |          |
        |          |         |             |          |          |
        |  Create  |         |             |          |          |
        |  Session |         |             |          |          |
    +   |<---------|         |             |          |          |
    |   |  Start   |         |             |          |          |
    |   | Handshake|         |             |          |          |
    |   |<---------|         |             |          |          |
    |   |   TLS    |         |             |          |          |
    |   | Records  |  Pack   |             |          |          |
    |   |--------->| Records |             |          |          |
        |          |-------->| send packet | Unpack   |          |
    R   |          |         |------------>| Records  | Create   |
    T   |          |         |             |--------->| Session  |
    T   |          |         |             |          |--------->|
        |          |         |             |          |   TLS    |
    1   |          |         |             |          | Records  |
        |          |         |             |          |--------->|
    |   |          |         |             |          |   TLS    |
    |   |          |         |             |  Pack    | Records  |
    |   |          |         |             | Records  |<---------|
    |   |          | Unpack  |send response|<---------|          |
    |   |   TLS    | Records |<------------|          |          |
    |   | Records  |<--------|             |          |          |
    +   |<---------|         |             |          |          |
        |   TLS    |         |             |          |          |
        | Records  |         |             |          |          |
    +   |--------->|-------->|------------>|--------->|--------->|
    |   |          |         |             |          |          |
        |          |         |             |          | Session  |
    R   |          |         |             |          |    Up    |
    T   |          |         |             |          |<---------|
    T   |          |         |             |          |   TLS    |
        |          |         |             |          | Records  |
    2   |<---------|<--------|<------------|<---------|<---------|
        | Session  |         |             |          |          |
    |   |    Up    |         |             |          |          |
    +   |--------->|         |             |          |          |
        |  Export  |         |             |          |  Export  |
        |   Keys   |         |             |          |   Keys   |
        |--------->|         | E2E Session |          |<---------|
        |          |<--------|-------------|--------->|          |
```

Figure 9: ATLS Session Establishment

9. ATLS over HTTP Transport

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 request and response bodies.

9.1. Protocol Summary

All ATLS records are transported unmodified as binary data within HTTP message bodies. The application simply extracts the TLS records from the TLS stack and inserts them directly into HTTP message bodies. Each message body contains a full TLS flight, which may contain multiple TLS records.

The client sends all ATLS records to the server in the bodies of POST requests.

The server sends all ATLS records to the client in the bodies of 200 OK responses to the POST requests.

9.2. Content-Type Header

A new Content-Type header value is defined:

Content-type: application/atls+octet-stream

All message bodies containing ATLS records must set this Content-Type. This enables middleboxes to readily identify ATLS payloads.

9.3. HTTP Status Codes

Thsi document does not define any new HTTP status codes, and does not specify additional semantics or refine existing semantics for status codes. This is the best current practice as outlined in [I-D.ietf-httpbis-bcp56bis].

9.4. ATLS Session Tracking

The application 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. The application service should use stateful cookies [RFC6265] in order to achieve this as recommended in [I-D.ietf-httpbis-bcp56bis].

9.5. Session Establishment and Key Exporting

It is recommended that applications using ATLS over HTTP transport only use ATLS for session establishment and key exchange, resulting in only 2 ATLS RTTs between the client and the application service.

Key exporting must be carried out as described in Section 7.3.

9.6. Application Data Encryption

[editors note: I am on the fence about using [RFC8188] as this hardcodes the ciphersuite to aes128gcm. It would be nice to use the cipher suite negotiated as part of ATLS session establishment.]

9.7. Illustrative ATLS over HTTP Session Establishment

A client initiates an ATLS session by sending the first TLS flight in a POST request message body to the ATLS server.

```
POST /atls
Content-Type: application/atls+octet-stream

<binary TLS client flight 1 records>
```

The server handles the request, creates an ATLS session object, and replies by including its first TLS flight in a 200 OK message body. The server also sets a suitable cookie for session tracking purposes.

```
200 OK
Content-Type: application/atls+octet-stream
Set-Cookie: my-atls-cookie=my-cookie-value

<binary TLS server flight 1 records>
```

The client handles the server first flight TLS records and replies with its second flight.

```
POST /atls
Content-Type: application/atls+octet-stream
Cookie: my-atls-cookie=my-cookie-value

<binary TLS client flight 2 records>
```

The server handles the second flight, establishes the ATLS session, and replies with its second flight.

```
200 OK
Content-Type: application/atls+octet-stream

<binary TLS server flight 2 records>
```

9.8. ATLS and HTTP CONNECT

It is worthwhile comparing and contrasting ATLS with HTTP CONNECT tunneling.

First, let us introduce some terminology:

- HTTP Proxy: A HTTP Proxy operates at the application layer, handles HTTP CONNECT messages from clients, and opens tunnels to remote origin servers on behalf of clients. If a client establishes a tunneled TLS connection to the origin server, the HTTP Proxy does not attempt to intercept or inspect the HTTP messages exchanged between the client and the server

- middlebox: A middlebox operates at the transport layer, terminates TLS connections from clients, and originates new TLS connections to services. A middlebox inspects all messages sent between clients and services. Middleboxes are generally completely transparent to applications, provided that the necessary PKI root Certificate Authority is installed in the client's trust store.

HTTP Proxies and middleboxes are logically separate entities and one or both of these may be deployed in a network.

HTTP CONNECT is used by clients to instruct a HTTP Forward Proxy deployed in the local domain to open up a tunnel to a remote origin server that is typically deployed in a different domain. Assuming that TLS transport is used between both client and proxy, and proxy and origin server, the network architecture is as illustrated in Figure 10. Once the proxy opens the transport tunnel to the service, the client establishes an end-to-end TLS session with the service, and the proxy is blindly transporting TLS records (the C->S TLS session records) between the client and the service. From the client perspective, it is tunneling a TLS session to the service inside the TLS session it has established to the proxy (the C->P TLS session). No middlebox is attempting to intercept or inspect the HTTP messages between the client and the service.

```
          +----------+        +----------+
          | C->S HTTP|        | C->S HTTP|
          +----------+        +----------+
          | C->S TLS |        | C->S TLS |
          +----------+        +----------+
          | C->P TLS |        | P->S TCP |
          +----------+        +----------+
          | C->P TCP |
          +----------+

   +--------+      +------------+      +---------+
   | Client |----->| HTTP Proxy |----->| Service |
   +--------+      +------------+      +---------+
```

Figure 10: HTTP Proxy transport layers

A more complex network topology where the network operator has both a HTTP Proxy and a middlebox deployed is illustrated in Figure 11. In this scenario, the proxy has tunneled the TLS session from the client towards the origin server, however the middlebox is intercepting and terminating this TLS session. A TLS session is established between the client and the middlebox (C->M TLS), and not end-to-end between the client and the server. It can clearly be seen that HTTP CONNECT and HTTP Proxies serve completely different functions than middleboxes.

Additionally, the fact that the TLS session is established between the client and the middlebox can be problematic for two reasons:

- o the middle box is inspecting traffic that is sent between the client and the service

- o the client may not have the necessary PKI root Certificate Authority installed that would enable it to validate the TLS connection to the middlebox. This is the scenario outlined in Section 3.1.

```
          +----------+        +----------+       +----------+
          | C->S HTTP|        | C->S HTTP|       | C->S HTTP|
          +----------+        +----------+       +----------+
          | C->M TLS |        | C->M TLS |       | M->S TLS |
          +----------+        +----------+       +----------+
          | C->P TLS |        | P->M TCP |       | M->S TCP |
          +----------+        +----------+       +----------+
          | C->P TCP |
          +----------+

   +--------+      +------------+      +-----------+      +---------+
   | Client |----->| HTTP Proxy |----->| Middlebox |----->| Service |
   +--------+      +------------+      +-----------+      +---------+
```

Figure 11: HTTP Proxy and middlebox transport layers

As HTTP CONNECT can be used to establish a tunneled TLS connection, one hypothetical solution to this middlebox issue is for the client to issue a HTTP CONNECT command to a HTTP Reverse Proxy deployed in front of the origin server. This solution is not practical for several reasons:

- if there is a local domain HTTP Forward Proxy deployed, this would result in the client doing a first HTTP CONNECT to get past the Forward Proxy, and then a second HTTP CONNECT to get past the Reverse Proxy. No client or client library supports the concept of HTTP CONNECT inside HTTP CONNECT.

- if there is no local domain HTTP Proxy deployed, the client still has to do a HTTP CONNECT to the HTTP Reverse Proxy. This breaks with standard and expected HTTP CONNECT operation, as HTTP CONNECT is only ever called if there is a local domain proxy.

- clients cannot generate CONNECT from XHR in web applications.

- this would require the deployment of a Reverse Proxy in front of the origin server, or else support of the HTTP CONNECT method in standard web frameworks. This is not an elegant design.

- using HTTP CONNECT with HTTP 1.1 to a Reverse Proxy will break middleboxes inspecting HTTP traffic, as the middlebox would see TLS records when it expects to see HTTP payloads.

In contrast to trying to force HTTP CONNECT to address a problem for which it was not designed to address, and having to address all the issues just outlined; ATLS is specifically designed to address the middlebox issue in a simple, easy to develop, and easy to deploy fashion.

- ATLS works seamlessly with HTTP Proxy deployments

- no changes are required to HTTP CONNECT semantics

- no changes are required to HTTP libraries or stacks

- no additional Reverse Proxy is required to be deployed in front of origin servers

It is also worth noting that if HTTP CONNECT to a Reverse Proxy were a conceptually sound solution, the solution still ultimately results in encrypted traffic traversing the middlebox that the middlebox cannot intercept and inspect. That is ultimately what ATLS results in - traffic traversing the middle box that the middlebox cannot intercept and inspect. Therefore, from a middlebox perspective, the differences between the two solutions are in the areas of solution complexity and protocol semantics. It is clear that ATLS is a simpler, more elegant solution that HTTP CONNECT.

10. ATLS over CoAP Transport

[todo: Help needed Hannes]

11. 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 software stack appropriately. Relevant configuration parameters for OpenSSL and Java SunJSSE stacks are outlined in the appendix.

12. IANA Considerations

[[TODO - New Content-Type and TLS Exporter Label must be registered.]]

13. Security Considerations

[[TODO]]

14. 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]]

15. 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 repository [[EDITOR'S NOTE: Add the URL here]].

15.1. 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);
```

15.2. 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!
```

16. 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]]

17. Informative References

[ALTS] Google, "Application Layer Transport Security", December 2017, <https://cloud.google.com/security/encryption-in-transit/application-layer-transport-security/>.

[Bluetooth] Bluetooth, "Bluetooth Core Specification v5.0", 2016, <https://www.bluetooth.com/>.

[I-D.hartke-core-e2e-security-reqs] Selander, G., Palombini, F., and K. Hartke, "Requirements for CoAP End-To-End Security", draft-hartke-core-e2e-security-reqs-03 (work in progress), July 2017.

[I-D.ietf-anima-bootstrapping-keyinfra] Pritikin, M., Richardson, M., Behringer, M., Bjarnason, S., and K. Watsen, "Bootstrapping Remote Secure Key Infrastructures (BRSKI)", draft-ietf-anima-bootstrapping-keyinfra-09 (work in progress), October 2017.

[I-D.ietf-core-object-security] Selander, G., Mattsson, J., Palombini, F., and L. Seitz, "Object Security for Constrained RESTful Environments (OSCORE)", draft-ietf-core-object-security-08 (work in progress), January 2018.

[I-D.ietf-httpbis-bcp56bis] Nottingham, M., "On the use of HTTP as a Substrate", draft-ietf-httpbis-bcp56bis-00 (work in progress), December 2017.

[I-D.ietf-tls-dtls13] Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", draft-ietf-tls-dtls13-22 (work in progress), November 2017.

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-23 (work in progress), January 2018.

[I-D.mattsson-core-security-overhead]
Mattsson, J., "Message Size Overhead of CoAP Security Protocols", draft-mattsson-core-security-overhead-02 (work in progress), November 2017.

[I-D.selander-ace-cose-ecdhe]
Selander, G., Mattsson, J., and F. Palombini, "Ephemeral Diffie-Hellman Over COSE (EDHOC)", draft-selander-ace-cose-ecdhe-07 (work in progress), July 2017.

[LwM2M] Open Mobile Alliance, "Lightweight Machine to Machine Requirements", December 2017, <http://www.openmobilealliance.org/>.

[Noise] Perrin, T., "Noise Protocol Framework", October 2017, <http://noiseprotocol.org/>.

[Norrell] Norrell, ., "Use SSL/TLS within a different protocol with BIO pairs", 2016, <https://thekerneldiaries.com/2016/06/13/openssl-ssltls-within-a-different-protocol/>.

[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>.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://www.rfc-editor.org/info/rfc6265>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <https://www.rfc-editor.org/info/rfc6347>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <https://www.rfc-editor.org/info/rfc7252>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

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

[Signal] Open Whisper Systems, "Signal Protocol", 2016, <https://signal.org/>.

[SSLEngine] Oracle, "SSLEngineSimpleDemo.java", 2004, <https://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/samples/sslengine/SSLEngineSimpleDemo.java>.

[ZigBee] ZigBee Alliance, "ZigBee Specification", 2012, <http://www.zigbee.org>.

Authors' Addresses

Owen Friel
Cisco

Email: ofriel@cisco.com

Richard Barnes
Cisco

Email: rlb@ipv.sx

Max Pritikin
Cisco

Email: pritikin@cisco.com

Hannes Tschofenig
ARM Limited

Email: hannes.tschofenig@gmx.net

Mark Baugher
Consultant

Email: mark@mbaugher.com