Network Working Group
Internet-Draft
Intended status: Informational
Expires: May 7, 2020

O. Friel
R. Barnes
M. Pritikin
Cisco
H. Tschofenig
Arm Ltd.
M. Baugher
Consultant

November 04, 2019

Application-Layer TLS
draft-friel-tls-atls-04

Abstract

This document specifies how TLS and DTLS can be used at the application layer for the purpose of establishing secure end-to-end encrypted communication security.

Encodings for carrying TLS and DTLS payloads are specified for HTTP and CoAP to improve interoperability. While the use of TLS and DTLS is straight forward we present multiple deployment scenarios to illustrate the need for end-to-end application layer encryption and the benefits of reusing a widely deployed and readily available protocol. 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 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 7, 2020.

Copyright Notice

Copyright (c) 2019 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 (https://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:

- 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.

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

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

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. TLS [RFC5246] [RFC8446] or DTLS [RFC6347] [I-D.ietf-tls-dtls13] can be used and this document is agnostic to the versions being used. 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

When TLS or DTLS is used at the application layer we refer to it as Application-Layer TLS, or ATLS. There is, however, no difference to TLS versions used over connection-oriented transports, such as TCP or SCTP. The same is true for DTLS. The difference is mainly in its use and the requirements placed on the underlying transport.

This document defines how ATLS can be used over HTTP [RFC7230] [RFC7540] and over CoAP [RFC7252]. This document does not preclude the use of other transports. However, defining how ATLS can be established over [ZigBee], [Bluetooth], etc. is beyond the scope of this document.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Application-Layer TLS is referred to as ATLS throughout this document.

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

This section describes describes a few end-to-end use cases in more detail.

3.1. Constrained Devices

Two constrained device use cases are outlined here.

3.1.1. Constrained Device Connecting over a Non-IP Network

There are industry examples of smart lighting systems where luminaires are connected using ZigBee to a gateway. A server connects to the gateway using CoAP over DTLS. The server can control the luminaires by sending messages and commands via the gateway. The gateway has full access to all messages sent between the luminaires and the server.

A generic use case similar to the smart lighting system outlined above has an IoT device talking ZigBee, Bluetooth Low Energy, LoRaWAN, NB-IoT, etc. to a gateway, with the gateway in turn talking CoAP over DTLS or another protocol to a server located in the cloud or on-premise. This is illustrated in Figure 1.

There are scenarios where certain messages sent between the IoT device and the server must not be exposed to the gateway function. Additionally, the two endpoints may not have 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.

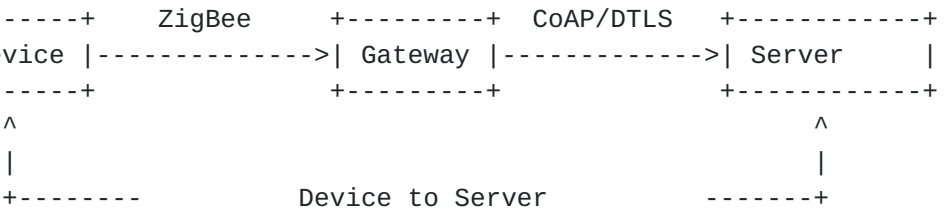


Figure 1: IoT Closed Network Gateway

3.1.2. Constrained Device Connecting over IP

In this example 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 2.

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.

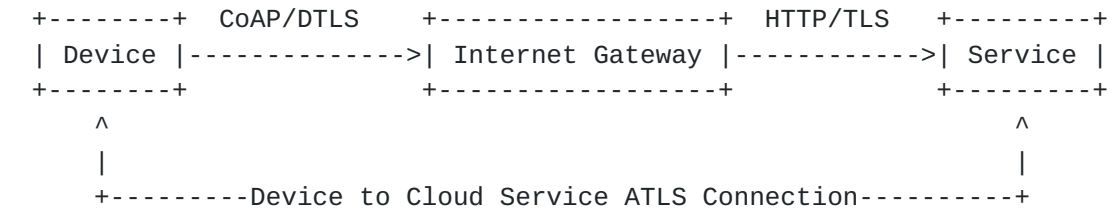


Figure 2: IoT Internet Gateway

3.2. 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 manage 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 3.

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 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.

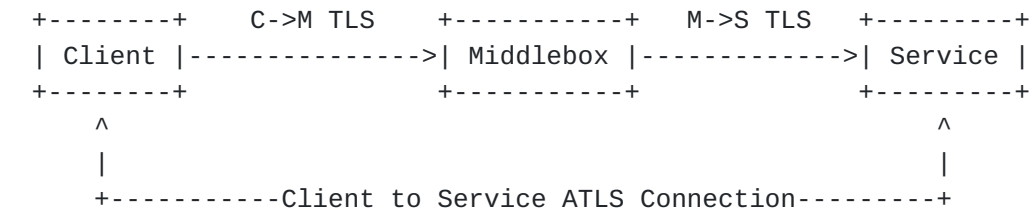


Figure 3: Bootstrapping Devices

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.

4. 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 TLS stacks and handshake protocols thus avoiding introducing new software or protocol dependencies in clients and applications
- reuse TLS [RFC5246] [RFC8446] and DTLS [RFC6347] [I-D.ietf-tls-dtls13] specifications,
- do not mandate constraints on how the TLS stack is configured or used,
- be forward compatible with future TLS versions including new developments such as compact TLS [I-D.rescorla-tls-ctls], and
- ensure that the design is as simple as possible.

5. Architecture Overview

5.1. 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 use ATLS, 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.

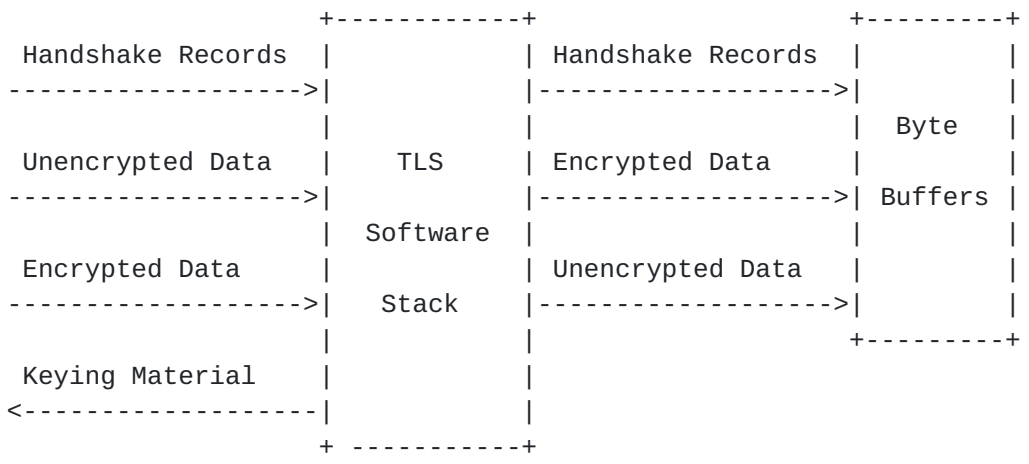


Figure 4: TLS Stack Interfaces

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.

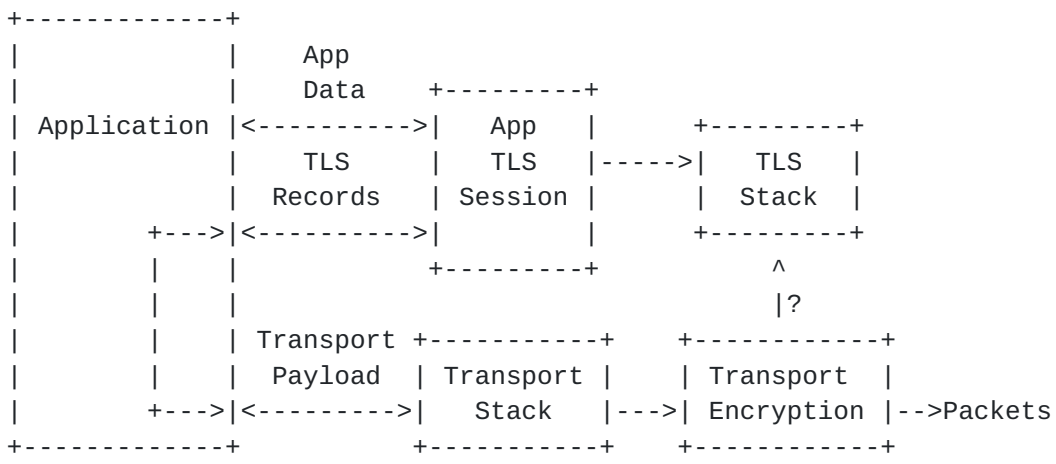


Figure 5: TLS Stack used for all data encryption

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.

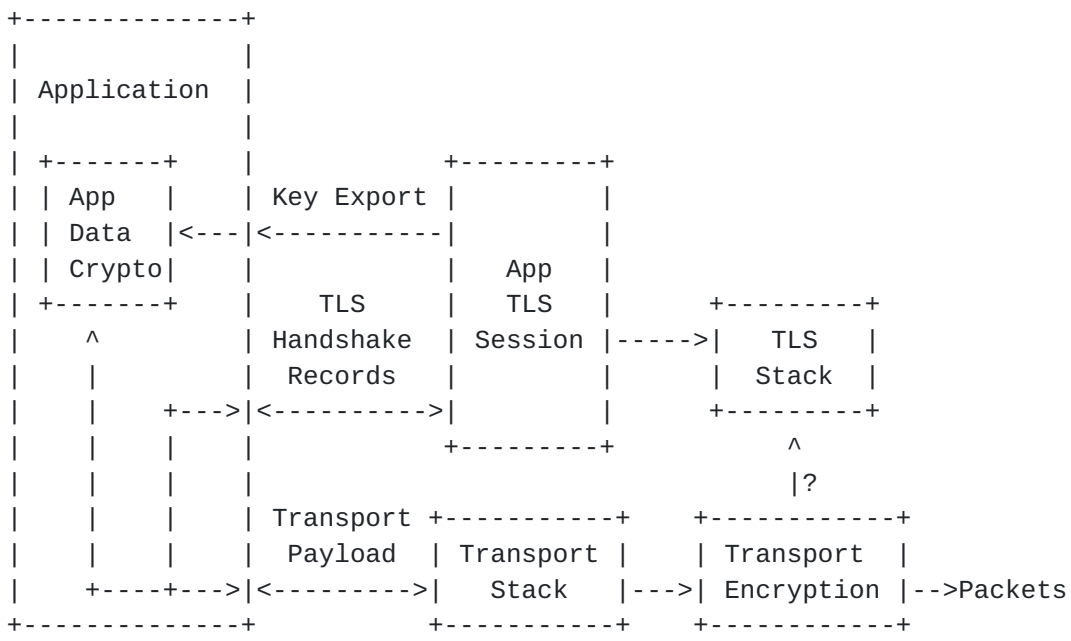


Figure 6: TLS stack used for key agreement and exporting

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.

- in some IoT use cases reducing the number of bytes transmitted is important. [I-D.mattsson-lwig-security-protocol-comparison] analyses the overhead of TLS headers compared with OSCORE [I-D.ietf-core-object-security] illustrating the additional overhead associated with TLS headers. The overhead varies between the different TLS versions and also between TLS and DTLS. 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.

5.1.1. 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

5.1.2. 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 identifying ATLS packets. The HTTP and CoAP encodings documented in Section 8 and Section 7 explicitly identify ATLS packets.

5.1.3. 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 8 and Section 7, but session tracking is application and implementation specific.

5.1.4. ATLS Record Inspection

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.

5.1.5. ATLS Message Routing

In many cases ATLS message routing is trival. However, there are potentially cases where the middlebox topology is quite complex and an example is shown in Figure 7. In this scenario multiple devices (Client 1-3) are connected using serial communication to a gateway (referred as middlebox A). Middlebox A communicates with another

middlebox B over UDP/IP. Middlebox B then interacts with some servers in the backend using CoAP over TCP.

This scenario raises the question about the ATLS message routing. In particular, there are two questions:

- How do the middleboxes know to which IP address to address the ATLS packet? This question arises in scenarios where clients are communicating over non-IP transports.

- How are response messages demultiplexed?

In some scenarios it is feasible to pre-configure the destination IP address of outgoing packets. Another other scenarios extra information available in the ATLS message or in a shim layer has to provide the necessary information. In the case of ATLS the use of the Server Name Indicating (SNI) parameter in the TLS/DTLS ClientHello message is a possibility to give middleboxes enough information to determine the ATLS communication endpoint. This approach is also compatible with SNI encryption.

For demultiplexing again different approaches are possible. The simplest approach is to use separate source ports for each ATLS session. In our example, Middlebox A allocates a dedicated socket (with a separate source port) for outgoing UDP datagrams in order to be able to relay a response message to the respective client. Alternatively, it is possible to make use of a shim layer on top of the transport that provides this extra demultiplexing capabilities. The use of multiple UDP "sessions" (as well as different TCP sessions) has the advantage of avoiding head-of-line blocking.

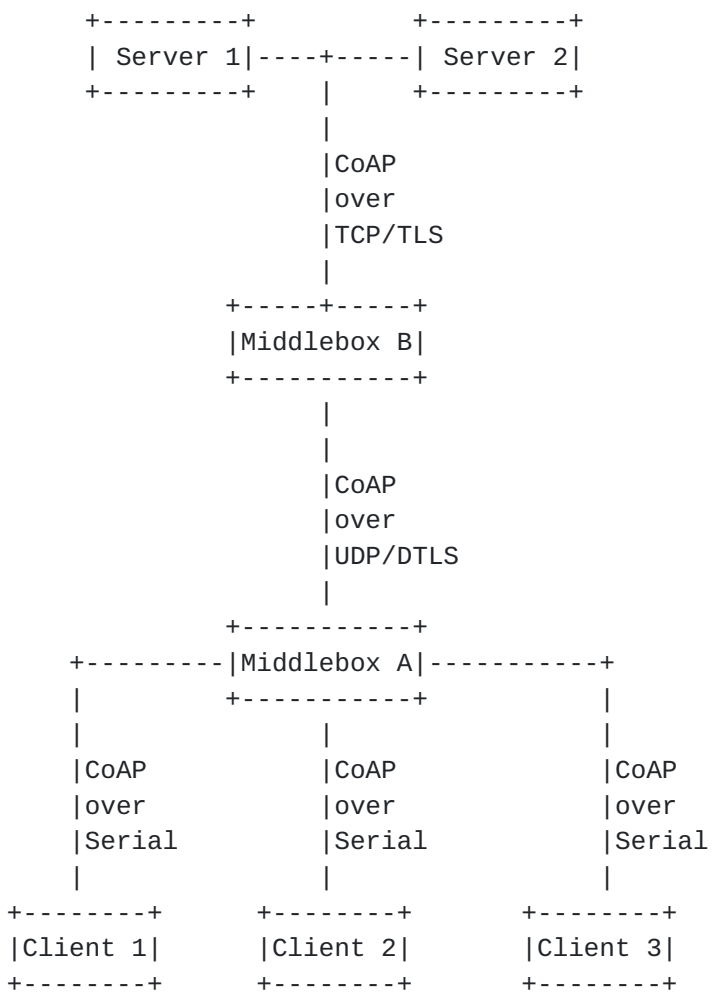


Figure 7: Message Routing Scenario

5.1.6. 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].

5.2. Functional Design

The functional design assumes that an authorization system has established operational keys for authenticating endpoints. In a layered design, this needs to be done for each layer, which may operate in two separate authorization domains. Note that Figure 8 shows a generic setup where TLS/DTLS is used at two layers. In some cases, use of TLS/DTLS at the application layer may be sufficient

where lower layer security mechanisms provide protection of the transport-specific headers.

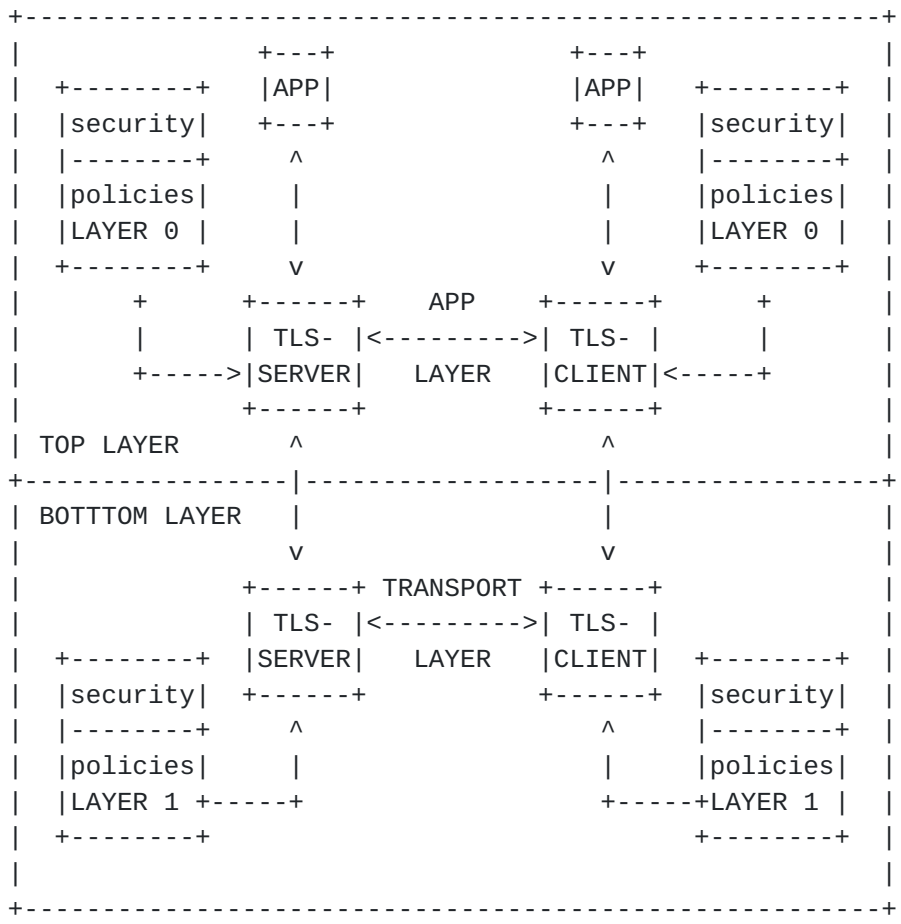


Figure 8: Functional Design

The security policies of one layer are distinct from those of another in Figure 8. They may overlap, but that is not necessary or perhaps even likely since the key exchanges at the different layers terminate at different endpoints and the two often have different authorization domains.

TLS can protect IoT device-to-gateway communications "on the wire" using the "bottom layer" of Figure 8, and it can protect application data from the device to the application server using the "top layer." Application and transport security each have a role to play. Transport security restricts access to messages on the networks, notably application headers and application-layer TLS restricts access to the application payloads.

As shown in Figure 8, an application-layer message, which gets encrypted and integrity protected and, in the generic case, the the

resulting TLS message and headers are passed to a TLS socket at the bottom layer, which may have a different security policy than the application layer.

5.3. Network Architecture

An example network deployment is illustrated in Figure 9. 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).

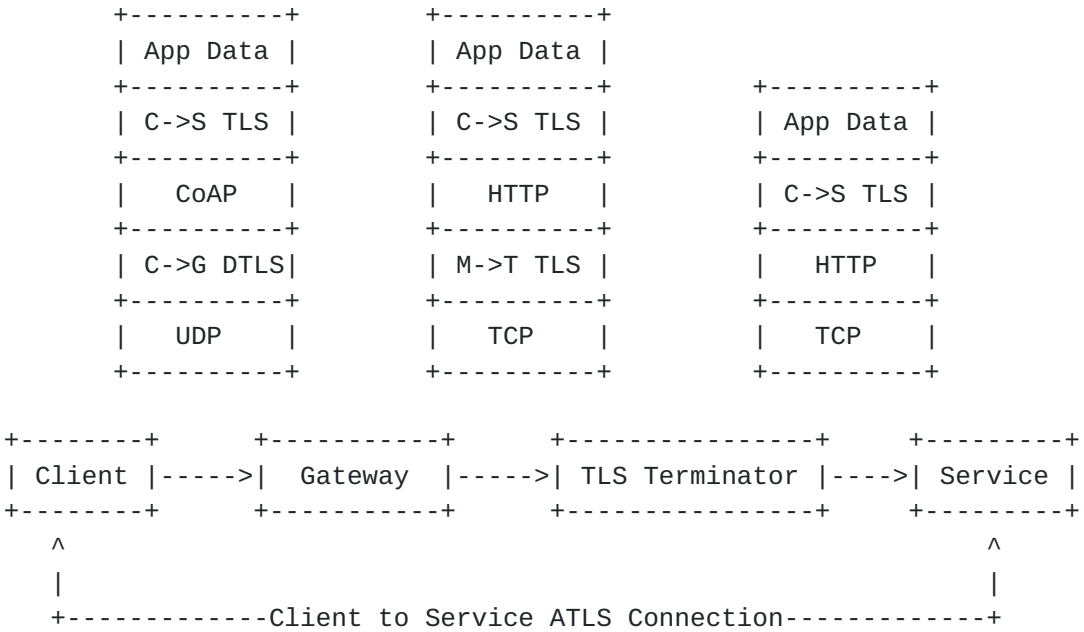


Figure 9: Constrained Device Gateway Network Architecture

Another typical network deployment is illustrated in Figure 10. 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).

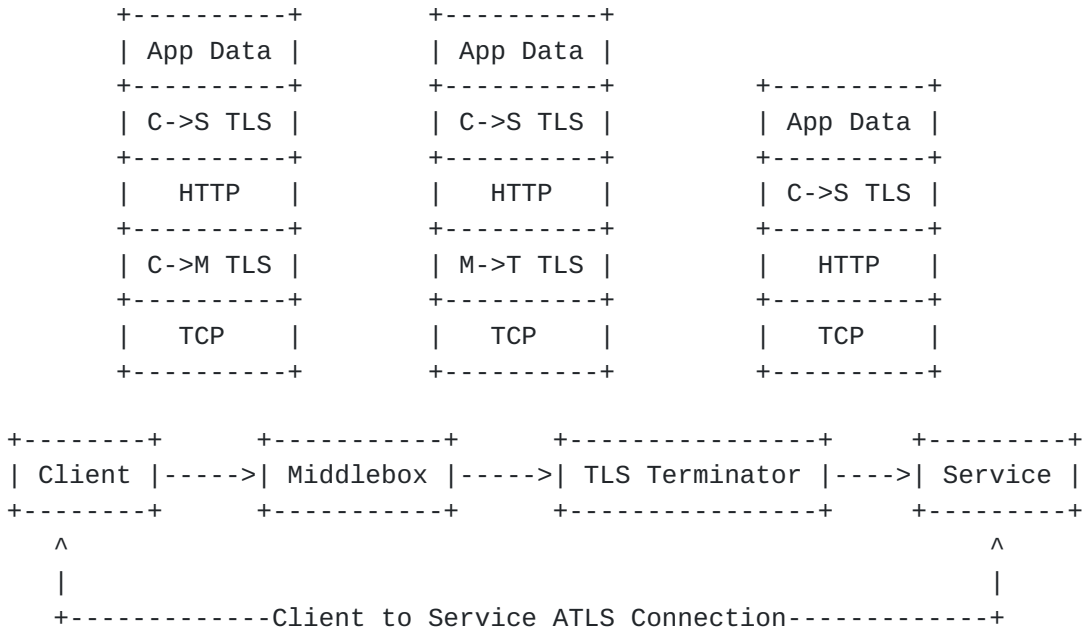


Figure 10: HTTP Middlebox Network Architecture

6. ATLS Session Establishment

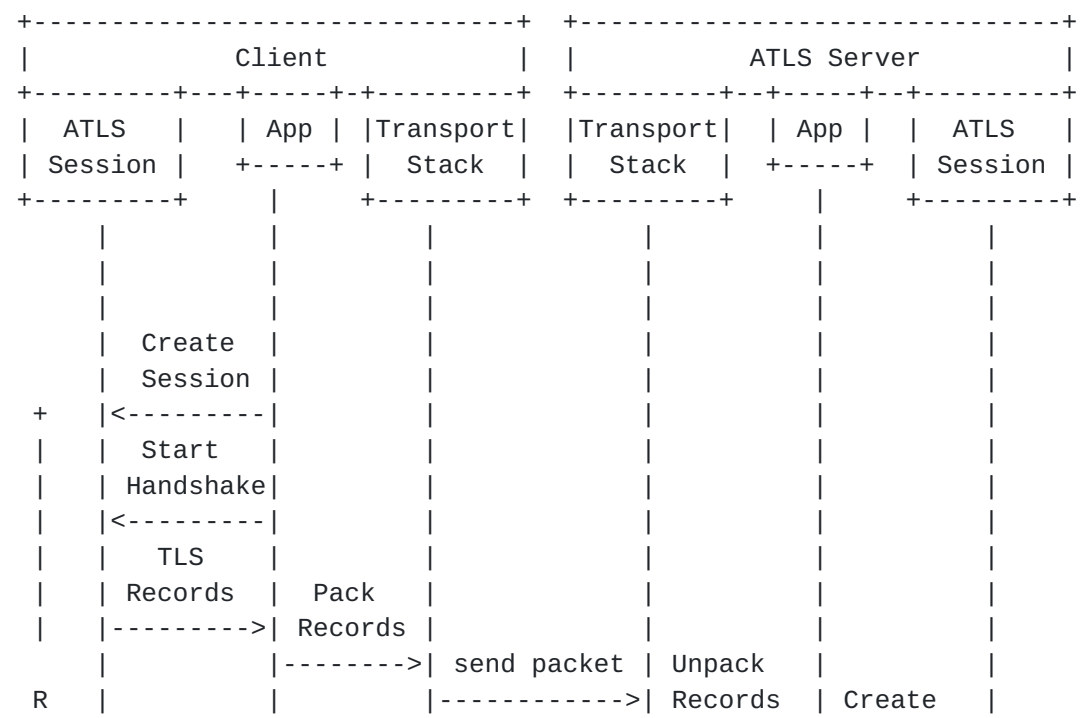
Figure 11 illustrates how an ATLS session is established using the key exporting architectural model shown in Figure 6. The number of RTTs that take place when establishing a TLS session depends on the version of TLS and what capabilities are enabled on the TLS software stack. For example, a 0-RTT exchange is possible with TLS 1.3. 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.

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



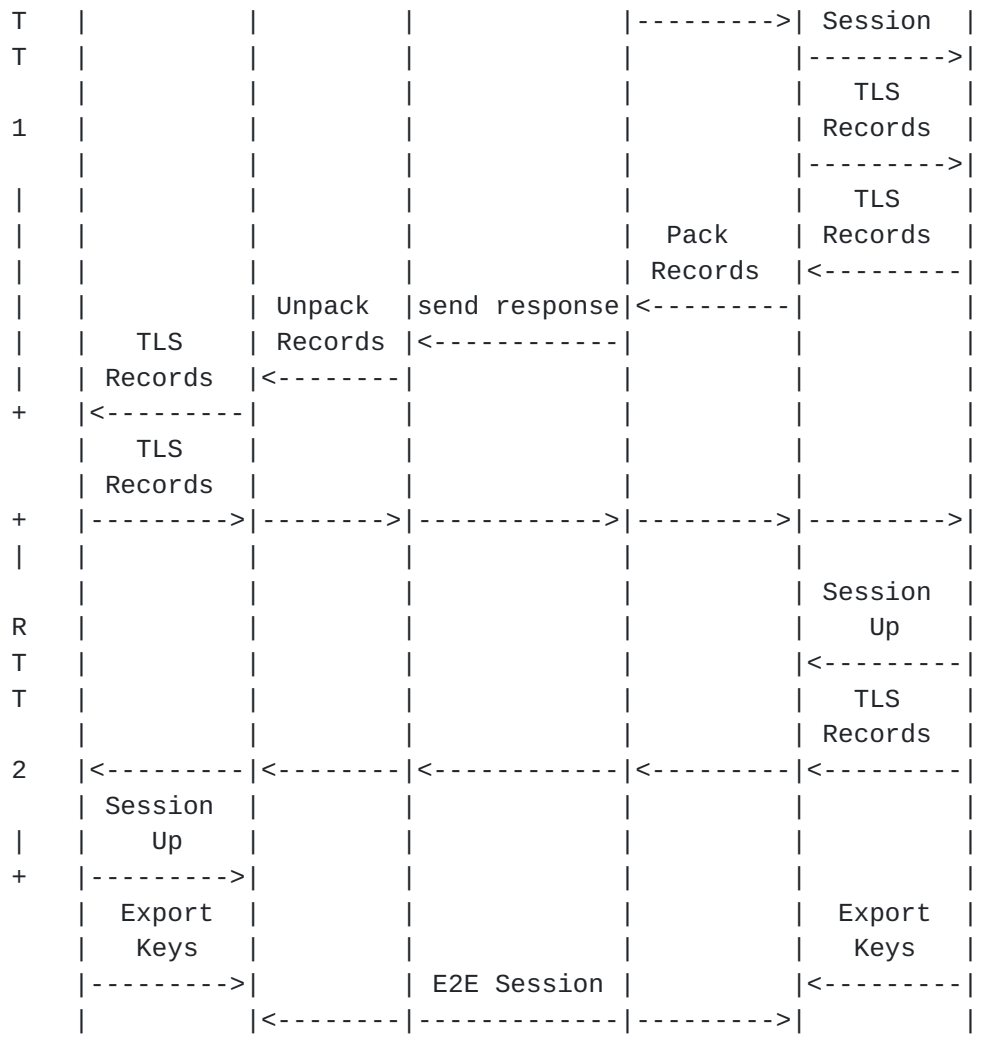


Figure 11: ATLS Session Establishment

7. ATLS over CoAP Transport

To carry TLS messages over CoAP [RFC7252] it is recommended to use Confirmable messages while DTLS payloads may as well use non-confirmable messages. The exchange pattern in CoAP uses the following style: A request from the CoAP client to the CoAP server uses a POST with the ATLS message contained in the payload of the request. An ATLS response is returned by the CoAP server to the CoAP client in a 2.04 (Changed) message.

When DTLS messages are conveyed in CoAP over UDP then the DDoS protection offered by DTLS MAY be used instead of replicating the functionality at the CoAP layer. If TLS is conveyed in CoAP over UDP then DDoS protection by CoAP has to be utilized. Carrying ATLS messages in CoAP over TCP does not require any additional DDoS protection.

The URI path used by ATLS is "/.well-known/atls".

{{coap-example} shows a TLS 1.3 handshake inside CoAP graphically.

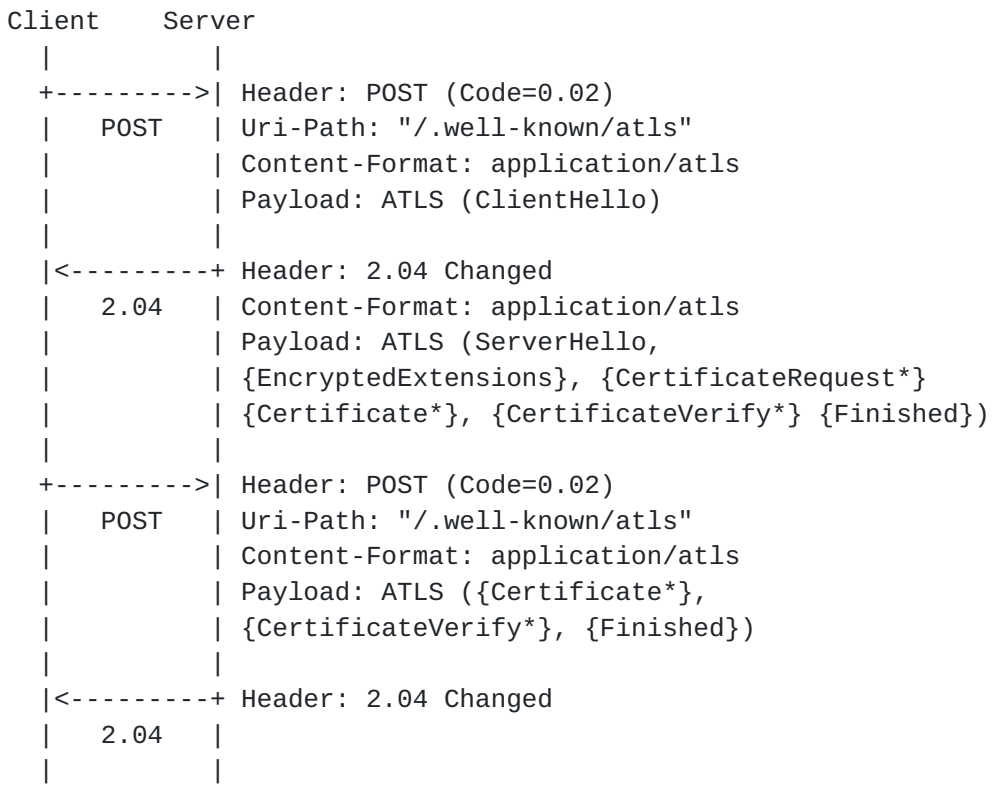


Figure 12: Transferring ATLS in CoAP

Note that application data can already be sent by the server in the second message and by the client in the third message, in case of the full TLS 1.3 handshake. In case of the 0-RTT handshake application data can be sent earlier. To mix different media types in the same CoAP payload the application/multipart-core content type is used.

Note also that CoAP blockwise transfer MAY be used if the payload size, for example due to the size of the certificate chain, exceeds the MTU size.

8. 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.

Note that ATLS over HTTP transport addresses a different deployment scenario than HTTP CONNECT proxies. HTTP CONNECT proxy behaviour is compared and contrasted with ATLS in Appendix B.

8.1. 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.

The URI path used by ATLS is "/.well-known/atls".

8.2. Content-Type Header

A new Content-Type header value is defined:

Content-type: application/atls

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

8.3. HTTP Status Codes

This 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].

8.4. ATLS Session Tracking

The application service needs to track multiple client application layer TLS sessions so that it can correlate 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].

8.5. 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 9.

8.6. 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 /.well-known/atls
Content-Type: application/atls

<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
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 /.well-known/atls
Content-Type: application/atls
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

<binary TLS server flight 2 records>
```

9. Key Exporting and Application Data Encryption

When solutions implement the architecture described in Figure 6, they leverage [RFC5705] for exporting keys. This section describes how to establish keying material and negotiate algorithms for OSCORE and for COSE.

9.1. OSCORE

When the OSCORE mode has been agreed using the "oscore_connection_id" extension defined in this document, different keys are used for DTLS/TLS record protection and for OSCORE packet protection. These keys are produced using a TLS exporter [RFC5705] and the exporter takes three input values:

- a disambiguating label string,
- a per-association context value provided by the application using the exporter, and
- a length value.

The label string for use with this specification is defined as 'atls-oscore'. The per-association context value is empty.

The length value is twice the size of the key size utilized by the negotiated algorithm since the lower-half is used for the Master Secret and the upper-half is used for the Master Salt.

For example, if a TLS/DTLS 1.2 handshake negotiated the TLS_PSK_WITH_AES_128_CCM_8 ciphersuite then the key size utilized by the negotiated algorithm, i.e. AES 128, is 128 bit. Hence, the key extractor is requested to produce 2 x 128 bit keying material.

The following parameters are needed for use with OSCORE:

- Master Secret: The master secret is derived as described above.
- Sender ID: This values is negotiated using the "oscore_connection_id" extension, as described in Section 11.1.
- Recipient ID: This values is negotiated using the "oscore_connection_id" extension, as described in Section 11.1.
- AEAD Algorithm: This value is negotiated using the ciphersuite exchange provided by the TLS/DTLS handshake. For example, if a TLS/DTLS 1.2 handshake negotiated the TLS_PSK_WITH_AES_128_CCM_8 ciphersuite then the AEAD algorithm identifier is AES_128_CCM_8,

which corresponds to two COSE algorithms, which both use AES-CCM mode with a 128-bit key, a 64-bit tag:

* AES-CCM-64-64-128

* AES-CCM-16-64-128 The difference between the two is only the length of the nonce, which is 7-bytes in the former case and 13-bytes in the latter. In TLS/DTLS the nonce value is not negotiated but fixed instead. Figure 13 provides the mapping between the TLS defined ciphersuite and the COSE algorithms.

- Master Salt: The master salt is derived as described above.

- HKDF Algorithm: This value is negotiated using the ciphersuite exchange provided by the TLS/DTLS handshake. As a default, SHA-256 is assumed as a HKDF algorithm for algorithms using 128-bit key sizes and SHA384 for 256-bit key sizes.

- Replay Window: A default window size of 32 packets is assumed.

9.2. COSE

The key exporting procedure for COSE is similiar to the one defined for OSCORE. The label string for use with this specification is defined as 'atls-cose'. The per-association context value is empty.

The length value is twice the size of the key size utilized by the negotiated algorithm since the lower-half is used for the Master Secret and the upper-half is used for the Master Salt.

The COSE algorithm corresponds to the ciphersuite negotiated during the TLS/DTLS handshake with with the mapping provided in Figure 13. The HKDF algorithm is negotiated using the the TLS/DTLS handshake. As a default, SHA-256 is assumed as a HKDF algorithm for algorithms using 128-bit key sizes and SHA384 for 256-bit key sizes.

COSE uses key ids to allow finding the appropriate security context. Those key IDs conceptually correspond to CIDs, as described in Section 11.2.

10. TLS Ciphersuite to COSE/OSCORE Algorithm Mapping

TLS Ciphersuite	COSE/OSCORE Algorithm
AES_128_CCM_8	AES-CCM w/128-bit key, 64-bit tag, 13-byte nonce
AES_256_CCM_8	AES-CCM w/256-bit key, 64-bit tag, 13-byte nonce
CHACHA20_POLY1305	ChaCha20/Poly1305 w/256-bit key, 128-bit tag
AES_128_CCM	AES-CCM w/128-bit key, 128-bit tag, 13-byte nonce
AES_256_CCM	AES-CCM w/256-bit key, 128-bit tag, 13-byte nonce
AES_128_GCM	AES-GCM w/128-bit key, 128-bit tag
AES_256_GCM	AES-GCM w/256-bit key, 128-bit tag

Figure 13: TLS Ciphersuite to COSE/OSCORE Algorithm Mapping

11. TLS Extensions

11.1. The "oscore_connection_id" Extension

This document defines the "oscore_connection_id" extension, which is used in ClientHello and ServerHello messages. It is used only for establishing the OSCORE Sender ID and the OSCORE Recipient ID. The OSCORE Sender ID maps to the CID provided by the server in the ServerHello and the OSCORE Recipient ID maps to the CID provided by the client in the ClientHello.

The negotiation mechanism follows the procedure used in [I-D.ietf-tls-dtls-connection-id] with the exception that the negotiated CIDs agreed with the "oscore_connection_id" extension is only used with OSCORE and does not impact the record layer format of the DTLS/TLS payloads nor the MAC calculation used by DTLS/TLS. As such, this extension can be used with DTLS as well as with TLS when those protocols are used at the application layer.

The extension type is specified as follows.

```
enum {
   oscore_connection_id(TBD), (65535)
} ExtensionType;

struct {
   opaque cid<0..2^8-1>;
} ConnectionId;
```

Figure 14: The 'oscore_connection_id' Extension

Note: This extension allows a client and a server to determine whether an OSCORE security context should be established.

11.2. The "cose_ext" Extension

This document defines the "cose_ext" extension, which is used in ClientHello and ServerHello messages. It is used only for establishing the key identifiers, AEAD algorithms, as well as keying material for use with application layer protection using COSE. The CID provided by the server in the ServerHello maps to the COSE kid transmitted from the client to the server and the CID provided by the client in the ClientHello maps to the COSE kid transmitted from the server to the client.

The negotiation mechanism follows the procedure used in [I-D.ietf-tls-dtls-connection-id] with the exception that the negotiated CIDs agreed with the "cose_ext" extension is only used with COSE and does not impact the record layer format of the DTLS/TLS payloads nor the MAC calculation used by DTLS/TLS. As such, this extension can be used with DTLS as well as with TLS when those protocols are used at the application layer.

The extension type is specified as follows.

```
enum {
   oscore_connection_id(TBD), (65535)
} ExtensionType;

struct {
   opaque cid<0..2^8-1>;
} ConnectionId;
```

Figure 15: The 'cose_ext' Extension

Note: This extension allows a client and a server to determine whether an COSE security context should be established.

12. IANA Considerations

12.1. "oscore_connection_id" TLS extension

IANA is requested to allocate two entries to the existing TLS "ExtensionType Values" registry, defined in [RFC5246], for oscore_connection_id(TBD1) and cose_ext(TBD2) defined in this document, as described in the table below.

Value	Extension Name	TLS 1.3	DTLS Only	Recommended	Reference
TBD1	oscore_connection_id	Y	N	N	[[This doc]]
TBD2	cose_ext	Y	N	N	[[This doc]]

Note: The "N" values in the Recommended column are set because these extensions are intended only for specific use cases.

12.2. TLS Ciphersuite to OSCORE/COSE Algorithm Mapping

IANA is requested to create a new registry for mapping TLS ciphersuites to SCORE/COSE algorithms

An initial mapping can be found in Figure 13.

Registration requests are evaluated after a three-week review period on the tls-reg-review@ietf.or mailing list, on the advice of one or more Designated Experts [RFC8126]. However, to allow for the allocation of values prior to publication, the Designated Experts may approve registration once they are satisfied that such a specification will be published.

Registration requests sent to the mailing list for review should use an appropriate subject (e.g., "Request to register an TLS - OSCORE/ COSE algorithm mapping: example"). Registration requests that are undetermined for a period longer than 21 days can be brought to the IESG's attention (using the iesg@ietf.org mailing list) for resolution.

Criteria that should be applied by the Designated Experts includes determining whether the proposed registration duplicates existing functionality, whether it is likely to be of general applicability or whether it is useful only for a single extension, and whether the registration description is clear.

IANA must only accept registry updates from the Designated Experts and should direct all requests for registration to the review mailing list.

12.3. .well-known URI Registry

IANA is requested to add the well-known URI 'atls' to the Well-Known URIs registry.

- URI suffix: atls
- Change controller: IETF
- Specification document(s): [[this document]]
- Related information: None

12.4. Media Types Registry

IANA is requested to add the media type 'application/atls' to the Media Types registry.

- Type name: application
- Subtype name: atls
- Required parameters: N/A
- Optional parameters: N/A
- Encoding considerations: binary
- Security considerations: See Security Considerations section of this document.
- Interoperability considerations: N/A
- Published specification: [[this document]] (this document)
- Applications that use this media type: Potentially any
- Fragment identifier considerations: N/A
- Additional information:
  - Magic number(s): N/A
  - File extension(s): N/A
  - Macintosh file type code(s): N/A
- Person & email address to contact for further information: See "Authors' Addresses" section.
- Intended usage: COMMON
- Restrictions on usage: N/A
- Author: See "Authors' Addresses" section.
- Change Controller: IESG

12.5. HTTP Content-Formats Registry

IANA is requested to add the media type 'application/atls' to the HTTP Content-Formats registry.

- Media Type: application/atls
- Encoding: binary
- ID: TBD
- Reference: [[this document]]

12.6. CoAP Content-Formats Registry

IANA is requested to add the media type 'application/atls' to the CoAP Content-Formats registry.

- Media Type: application/atls
- Encoding: binary
- ID: TBD
- Reference: [[this document]]

12.7. TLS Key Extractor Label

IANA is requested to register the "application-layer-tls" label in the TLS Extractor Label Registry to correspond to this specification.

13. Security Considerations

This specification re-uses the TLS and DTLS and hence the security considerations of the respective TLS/DTLS version applies. As described in Section 5.2, implementers need to take the policy configuration into account when applying security protection at various layers of the stack even if the same protocol is used since the communiation endpoints and the security requirements are likely going to vary.

For use in the IoT environment the considerations described in [RFC7925] apply and other environments the guidelines in [RFC7525] are applicable.

14. References

14.1. Normative References

[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-16 (work in progress), March 2019.

[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-dtls13-33 (work in progress), October 2019.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[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>.

[RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <https://www.rfc-editor.org/info/rfc7525>.

[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>.

[RFC7925] Tschofenig, H., Ed. and T. Fossati, "Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things", RFC 7925, DOI 10.17487/RFC7925, July 2016, <https://www.rfc-editor.org/info/rfc7925>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

14.2. 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.ietf-anima-bootstrapping-keyinfra] Pritikin, M., Richardson, M., Eckert, T., Behringer, M., and K. Watsen, "Bootstrapping Remote Secure Key Infrastructures (BRSKI)", draft-ietf-anima-bootstrapping-keyinfra-29 (work in progress), October 2019.

[I-D.ietf-httpbis-bcp56bis]
Nottingham, M., "Building Protocols with HTTP", draft-ietf-httpbis-bcp56bis-09 (work in progress), November 2019.

[I-D.ietf-tls-dtls-connection-id]
Rescorla, E., Tschofenig, H., and T. Fossati, "Connection Identifiers for DTLS 1.2", draft-ietf-tls-dtls-connection-id-07 (work in progress), October 2019.

[I-D.mattsson-lwig-security-protocol-comparison]
Mattsson, J. and F. Palombini, "Comparison of CoAP Security Protocols", draft-mattsson-lwig-security-protocol-comparison-01 (work in progress), March 2018.

[I-D.rescorla-tls-ctls]
Rescorla, E. and R. Barnes, "Compact TLS 1.3", draft-rescorla-tls-ctls-02 (work in progress), July 2019.

[I-D.selander-ace-cose-ecdhe]
Selander, G., Mattsson, J., and F. Palombini, "Ephemeral Diffie-Hellman Over COSE (EDHOC)", draft-selander-ace-cose-ecdhe-14 (work in progress), September 2019.

[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/>.

[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>.

Appendix A. 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 note that this is illustrative, non-functional pseudo code that does not compile.

A.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);
```

A.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!
```

Appendix B. 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 16. 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.

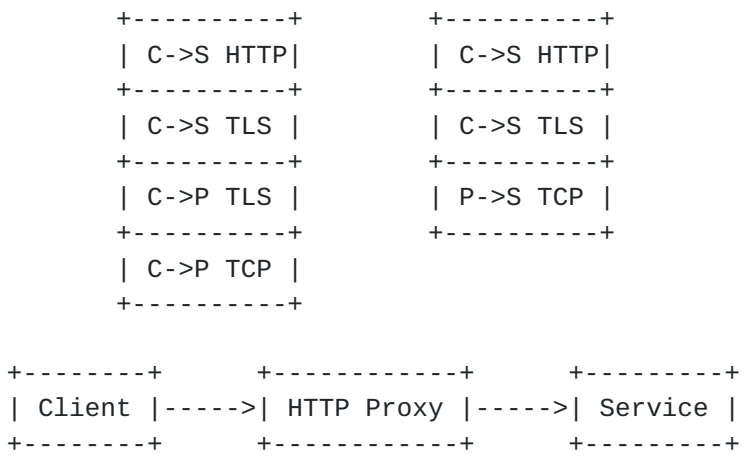


Figure 16: HTTP Proxy transport layers

A more complex network topology where the network operator has both a HTTP Proxy and a middlebox deployed is illustrated in Figure 17. 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:

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

- 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 3.2.

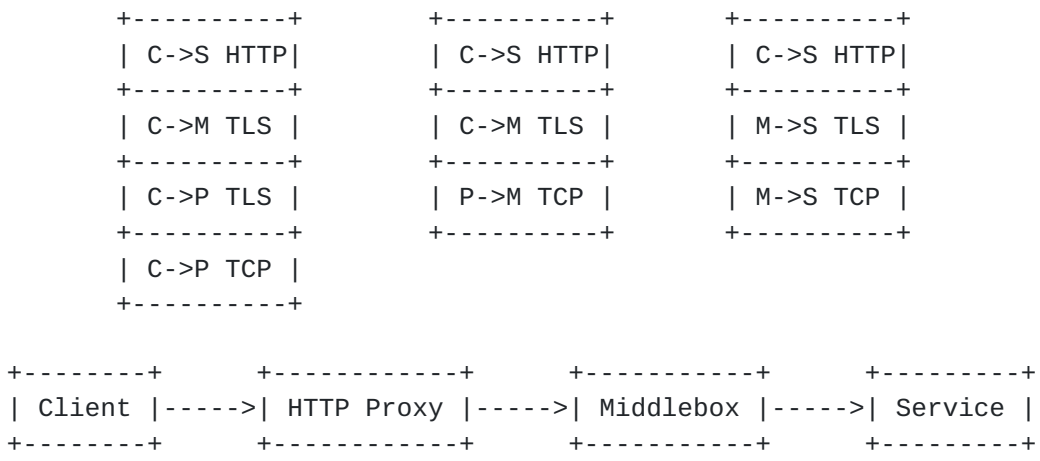


Figure 17: HTTP Proxy and middlebox transport layers

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.

Appendix C. Alternative 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

C.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 ~~~

C.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

C.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.

C.4. Ephemeral Diffie-Hellman Over COSE (EDHOC)

There is ongoing work to standardise EDHOC [I-D.selander-ace-cose-ecdhe], which defines a SIGMA-I based authenticated key exchange protocol using COSE and CBOR.

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 Ltd.

Email: hannes.tschofenig@gmx.net

Mark Baugher
Consultant

Email: mark@mbaugher.com