TCPING
Internet-Draft
Intended status: Standards Track
Expires: April 4, 2016

E. Rescorla
Mozilla

October 2, 2015

# TCP Use TLS Option
# draft-rescorla-tcpinc-tls-option-04

## Abstract

This document defines the use of TLS [RFC5246] with the TCP-ENO option [I-D.bittau-tcpinc-tcpeno].

## Status of This Memo

## Copyright Notice

Table of Contents

## 1. Introduction

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH The source for this draft is maintained in GitHub. Suggested changes should be submitted as pull requests at https://github.com/ekr/tcpinc-tls. Instructions are on that page as well.

The TCPINC WG is chartered to define protocols to provide ubiquitous, transparent security for TCP connections. The WG is specifying The TCP Encryption Negotiation Option (TCP-ENO) [I-D.bittau-tcpinc-tcpeno] which allows for negotiation of encryption at the TCP layer. This document describes a binding of TLS [RFC5246] to TCP-ENO as what ENO calls an "encryption spec", thus allowing TCP-ENO to negotiate TLS.

## 2. Overview

The basic idea behind this draft is simple. The SYN and SYN/ACK messages carry the TCP-ENO options indicating the willingness to do TLS. If both sides want to do TLS, then a TLS handshake is started and once that completes, the data is TLS protected prior to being sent over TCP. Otherwise, the application data is sent as usual.

```
Client                                   Server

SYN + TCP-ENO [TLS]->
                     <- SYN/ACK + TCP-ENO [ENO]
ACK ->
<---------------- TLS Handshake --------------->
<--------- Application Data over TLS ---------->
```

Figure 1: Negotiating TLS with TCP-TLS

```
Client                                   Server

SYN + TCP-ENO [TLS] ->
                                      <- SYN/ACK
ACK ->
<--------- Application Data over TLS ---------->
```

Figure 2: Fall back to TCP

If use of TLS is negotiated, the data sent over TCP simply is TLS data in compliance with TLS 1.2 [RFC5246] or TLS 1.3 [I-D.ietf-tls-tls13].

Once the TLS handshake has completed, all application data SHALL be sent over that negotiated TLS channel. Application data MUST NOT be sent prior to the TLS handshake.

If the TLS handshake fails for non-cryptographic reasons such as failure to negotiate a compatible cipher or the like, endpoints SHOULD behave as if the the TCP-TLS option was not present. This is obviously not the conventional behavior for TLS failure, but as the entire idea here is to be opportunistic and the attacker can simply suppress the TCP-TLS option entirely, this provides the maximum robustness against broken intermediaries. If the TLS handshake fails for cryptographic reasons that indicate damage to the datastream (e.g., a decryption failure or a Finished failure) then the endpoints SHOULD signal a connection failure, as this suggests that there is a middlebox modifying the data and there is a reasonable chance that the state is now corrupted.

## 3. TLS Profile

The TLS Profile defined in this document is intended to be a compromise between two separate use cases. For the straight TCPINC use case of ubiquitous transport encryption, we desire that implementations solely implement TLS 1.3 [I-D.ietf-tls-tls13] or greater. However, we also want to allow the use of TCP-ENO as a

signal for applications to do out-of-band negotiation of TLS, and those applications are likely to already have support for TLS 1.2 [RFC5246]. In order to accomodate both cases, we specify a wire encoding that allows for negotiation of multiple TLS versions (Section 4) but encourage implementations to implement only TLS 1.3. Implementations which also implement TLS 1.2 MUST implement the profile described in Section 3.2

## 3.1. TLS 1.3 Profile

TLS 1.3 is the preferred version of TLS for this specification. In order to facilitate implementation, this section provides a non-normative description of the parts of TLS 1.3 which are relevant to TCPINC and defines a baseline of algorithms and modes which MUST be supported. Other modes, cipher suites, key exchange algorithms, certificate formats as defined in [I-D.ietf-tls-tls13] MAY also be used and that document remains the normative reference for TLS 1.3. Bracketed references (e.g., [S. 1.2.3.4] refer to the corresponding section in that document.) In order to match TLS terminology, we use the term "client" to indicate the TCP-ENO "A" role (See [I-D.bittau-tcpinc-tcpeno]; Section 3.1) and "server" to indicate the "B" role.

### 3.1.1. Handshake Modes

TLS 1.3 as used in TCPINC supports two handshake modes, both based on ECDHE key exchange.

- A 1-RTT mode which is used when the client has no information about the server's keying material (see Figure 1)

- A 0-RTT mode which is used when the client and server have connected previous and which allows the client to send data on the first flight (see Figure 2

In both case, the server is expected to have an ECDSA signing key which may either be a freshly-generated key or a long-term key (allowing TOFU-style applications). The key need not be associated with any certificate and can simply be a bare key.

Full TLS 1.3 includes support for additional modes based on pre-shared keys, but TCPINC implementations MAY opt to omit them. Implementations MUST implement the 1-RTT mode and SHOULD implement the 0-RTT mode.

```
       Client                                               Server

       ClientHello
         + ClientKeyShare        -------->
                                                       ServerHello
                                                    ServerKeyShare
                                             {EncryptedExtensions}
                                            {ServerConfiguration*}
                                                     {Certificate}
                                               {CertificateVerify}
                                 <--------              {Finished}
       {Finished}                -------->
       [Application Data]        <------->      [Application Data]

              *  Indicates optional or situation-dependent
                 messages that are not always sent.

              {} Indicates messages protected using keys
                 derived from the ephemeral secret.

              [] Indicates messages protected using keys
                 derived from the master secret.
```

Figure 1: Message flow for full TLS Handshake

Note: Although these diagrams indicate a message called "Certificate", this message MAY either contain a bare public key or an X.509 certificate (this is intended to support the out-of-band use case indicated above).  Implementations MUST support bare public keys and MAY support X.509 certificates.

### 3.1.2. Basic 1-RTT Handshake

#### 3.1.2.1. Client's First Flight

##### 3.1.2.1.1. Sending

In order to initiate the TLS handshake, the client sends a "ClientHello" message [S. 6.3.1.1].

```
    struct {
        ProtocolVersion client_version = { 3, 4 };    /* TLS v1.3 */
        Random random;
        uint8 session_id_len_RESERVED;                /* Must be zero */
        CipherSuite cipher_suites<2..2^16-2>;
        uint8 compression_methods_len_RESERVED;       /* Must be zero */
        Extension extensions<0..2^16-1>;
    } ClientHello;
```

The fields listed here have the following meanings:

client_version
   The version of the TLS protocol by which the client wishes to communicate during this session.

random
   A 32-byte random nonce.

cipher_suites
   This is a list of the cryptographic options supported by the client, with the client's first preference first.

extensions contains a set of extension fields.  The client MUST include the following extensions:

SignatureAlgorithms [S. 6.3.2.1]
   A list of signature/hash algorithm pairs the client supports.

NamedGroup [S. 6.3.2.2]
   A list of ECDHE groups that the client supports

ClientKeyShare [S. 6.3.2.3]
   Zero or more ECDHE shares drawn from the groups in NamedGroup. This SHOULD contain either a P-256 key or an X25519 key.

The client MUST also include a ServerCertTypeExtension containing type "Raw Public Key" [RFC7250], indicating its willingness to accept a raw public key rather than an X.509 certificate in the server's Certificate message.

#### 3.1.2.1.2.  Receiving

Upon receiving the client's ClientHello, the server selects a ciphersuite and ECDHE group out of the lists provided by the client in the cipher_suites list and the NamedGroup extension.  If the client supplied an appropriate ClientKeyShare for that group, then the server responds with a ServerHello (see {{server-first-flight). Otherwise, it replies with a HelloRetryRequest (Section 3.1.3), indicating that the client needs to re-send the ClientHello with an appropriate key share; because all TCPINC implementations are required to support P-256, this should not happen unless P-256 is deprecated by a subsequent specification.

#### 3.1.2.2. Server's First Flight

##### 3.1.2.2.1. Sending

The server respond's to the client's first flight with a sequence of messages:

ServerHello [6.3.1.2]
Contains a nonce and the cipher suite that the server has selected out of the client's list. The server MUST support the extensions listed in Section 3.1.2.1.1 and MUST also ignore any extensions it does not recognize; this implies that the server can implement solely the extensions listed in Section 3.1.2.1.1.

ServerKeyShare [6.3.3]
Contains the server's ECDHE share for one of the groups offered in the client's ClientKeyShare message. All messages after ServerKeyShare are encrypted using keys derived from the ClientKeyShare and ServerKeyShare.

EncryptedExtensions [6.3.4]
Responses to the extensions offered by the client. In this case, the only relevant extension is the ServerCertTypeExtension.

Certificate [6.3.5]
The server's certificate. If the client offered a "Raw Public Key" type in ServerCertTypeExtension this message SHALL contain a SubjectPublicKeyInfo value for the server's key as specified in [RFC7250]. Otherwise, it SHALL contain one or more X.509 Certificates, as specified in [I-D.ietf-tls-tls13], Section 6.3.5. In either case, this message MUST contain a key which is consistent with the client's SignatureAlgorithms and NamedGroup extensions.

ServerConfiguration [6.3.7]
A server configuration value for use in 0-RTT (see Section 3.1.4).

CertificateVerify [6.3.8]
A signature over the handshake transcript using the key provided in the certificate message.

Finished [6.3.9]
A MAC over the entire handshake transcript up to this point.

Once the server has sent the Finished message, it can immediately generate the application traffic keys and start sending application traffic to the client.

#### 3.1.2.3. Receiving

Upon receiving the server's first flight, the client proceeds as follows:

- o Read the ServerHello message to determine the cryptographic parameters.

- o Read the ServerKeyShare message and use that in combination with the ClientKeyShare to compute the keys which are used to encrypt the rest of the handshake.

- o Read the EncryptedExtensions message. As noted above, the main extension which needs to be processed is ServerCertTypeExtension, which indicates the format of the server's certificate message.

- o Read the server's certificate message and store the server's public key. Unless the implementation is specifically configured otherwise, it SHOULD NOT attempt to validate the certificate, even if it is of type X.509 but merely extract the key.

- o Read the server's CertificateVerify message and verify the server's signature over the handshake transcript. If the signature does not verify, the client terminates the handshake with an alert (Section 6.1.2).

- o Read the server's Finished message and verify the finished MAC based on the DH shared secret. If the MAC does not verify, the client terminates the handshake with an alert.

#### 3.1.2.4. Client's Second Flight

Finally, the client sends a Finished message which contains a MAC over the handshake transcript (except for the server's Finished). [[TODO: In the upcoming draft of TLS 1.3, the client's Finished will likely include the server's Finished.]] Once the client has transmitted the Finished, it can begin sending encrypted traffic to the server.

The server reads the client's Finished message and verifies the MAC. If the MAC does not verify, the client terminates the handshake with an alert.

### 3.1.3. Hello Retry Request [6.3.1.3]

Because there are a small number of recommended groups, the ClientKeyShare will generally contain a key share for a group that the server supports. However, it is possible that the client will

not send such a key share, but there may be another group that the client and server jointly support. In that case, the server MUST send a HelloRetryRequest indicating the desired group:

```
      struct {
          ProtocolVersion server_version;
          CipherSuite cipher_suite;
          NamedGroup selected_group;
          Extension extensions<0..2^16-1>;
      } HelloRetryRequest;
```

In response to the HelloRetryRequest the client re-sends its ClientHello but with the addition of the group indicated in "selected_group".

### 3.1.4. Zero-RTT Exchange

TLS 1.3 allows the server to send its first application data message to the client immediately upon receiving the client's first handshake message (which the client can send upon receiving the server's SYN/ ACK). However, in the basic handshake, the client is required to wait for the server's first flight before it can send to the server. TLS 1.3 also includes a "Zero-RTT" feature which allows the client to send data on its first flight to the server.

In order to enable this feature, in an initial handshake the server sends a ServerConfiguration message which contains the server's semi-static (EC)DH key which can be used for a future handshake:

```
      struct {
          opaque configuration_id<1..2^16-1>;
          uint32 expiration_date;
          NamedGroup group;
          opaque server_key<1..2^16-1>;
          EarlyDataType early_data_type;
          ConfigurationExtension extensions<0..2^16-1>;
      } ServerConfiguration;
```

The group and server_key fields contain the server's (EC)DH key and the early_data_type field is used to indicate what data can be sent in zero-RTT. Because client authentication is forbidden in TCPINC-uses of TLS 1.3 (see Section 3.3), the only valid value here is "early_data", indicating that the client can send data in 0-RTT.

When a ServerConfiguration is available, the client can send an EarlyDataIndication extension in its ClientHello and then start sending data immediately, as shown below.

```
          Client                                               Server

          ClientHello
            + ClientKeyShare
            + EarlyDataIndication
          (EncryptedExtensions)
          (Application Data)        -------->
                                                          ServerHello
                                                + EarlyDataIndication
                                                       ServerKeyShare
                                                {EncryptedExtensions}
                                               {ServerConfiguration*}
                                                        {Certificate}
                                                  {CertificateVerify}
                                    <--------              {Finished}
          {Finished}                -------->

          [Application Data]        <------->      [Application Data]

               () Indicates messages protected using keys
                  derived from the static secret.
```

Figure 2: Message flow for a zero round trip handshake

IMPORTANT NOTE: TLS 1.3 Zero-RTT data is inherently replayable (see the note in [I-D.ietf-tls-tls13] Section 6.2.2). If only passive threat models are relevant, this issue becomes less important. However, if applications are performing an external channel binding using the session id to prevent active attack, then care must be taken to prevent this form of attack. See Section 6.2.2 of [I-D.ietf-tls-tls13] for more information on this topic. [[OPEN ISSUE: can we use data from the TCP SYN as anti-replay stuff.]]

### 3.1.5. Key Schedule

TLS 1.3 derives its traffic keys from two input keying material values:

Ephemeral Secret (ES): A secret which is derived from ClientKeyShare and ServerKeyShare.

Static Secret (SS): A secret which which is derived from ClientKeyShare and either ServerKeyShare (in the 1-RTT case) or the public key in the ServerConfiguration (in the 0-RTT case).

The handshake is encrypted under keys derived from ES. The ordinary traffic keys are derived from the combination of ES and SS. The 0-RTT traffic keys are derived solely from ES and therefore have

limited forward security. All key derivation is done using HKDF [RFC5869].

### 3.1.6. Record Protection

Once the TLS handshake has completed, all data is protected as a series of TLS Records.

```
       struct {
           ContentType opaque_type = application_data(23); /* see fragment.type
*/
           ProtocolVersion record_version = { 3, 1 };    /* TLS v1.x */
           uint16 length;
           aead-ciphered struct {
              opaque content[TLSPlaintext.length];
              ContentType type;
              uint8 zeros[length_of_padding];
           } fragment;
       } TLSCiphertext;
```

Each record is encrypted with an AEAD cipher with the following parameters:

- The AEAD nonce is constructed by generating a per-connection nonce mask of length max(8 bytes, N_MIN) for the AEAD algorithm (see [RFC5116] Section 4) and XORing it with the record sequence number (left-padded with zeroed).

- The additional data is the sequence number + the TLS version number.

The record data MAY BE padded with zeros to the right. Because the content type value is always non-zero, the padding is removed by removing bytes from the right until a non-zero byte is encountered.

### 3.2. TLS 1.2 Profile

Implementations MUST implement and require the TLS Extended Master Secret Extension [I-D.ietf-tls-session-hash] and MUST NOT negotiate versions of TLS prior to TLS 1.2. Implementations MUST NOT negotiate non-AEAD cipher suites and MUST use only PFS cipher suites with a key of at least 2048 bits (finite field) or 256 bites (elliptic curve). TLS 1.2 implementations MUST NOT initiate renegotiation and MUST respond to renegotiation with a fatal "no_renegotiation" alert.

### 3.3. Deprecated Features

When TLS is used with TCPINC, a number of TLS features MUST NOT be used, including:

- TLS certificate-based client authentication
- Session resumption [????]

### 3.4. Session ID

TCP-ENO Section 4.1 defines a session ID feature (not to be confused with TLS Session IDs). When the protocol in use is TLS, the session ID is computed via a TLS Exporter [RFC5705] using the Exporter Label [[TBD]] and with the "context" input being the TCP-ENO negotiation transcript defined in [I-D.bittau-tcpinc-tcpeno] Section 3.4.

### 3.5. Cryptographic Algorithms

Implementations of this specification MUST implement the following cipher suite:

```
    TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
```

These cipher suites MUST support both digital signatures and key exchange with secp256r1 (NIST P-256) and SHOULD support key agrement with X25519 [I-D.irtf-cfrg-curves].

Implementations of this specification SHOULD implement the following cipher suites:

```
    TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305
    TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
```

## 4. Suboption Definition

This document uses a one byte TCP-ENO suboption. See Section 8.

## 5. Transport Integrity

The basic operational mode defined by TCP-TLS protects only the application layer content, but not the TCP segment metadata. Upon receiving a packet, implementations MUST first check the TCP checksum and discard corrupt packets without presenting them to TLS. If the TCP checksum passes but TLS integrity fails, the connection MUST be torn down.

Thus, TCP-TLS provides automatic security for the content, but not protection against DoS-style attacks. For instance, attackers will be able to inject RST packets, bogus application segments, etc., regardless of whether TLS authentication is used. Because the application data is TLS protected, this will not result in the application receiving bogus data, but it will constitute a DoS on the connection.

This attack could be countered by using TCP-TLS in combination with TCP-AO [RFC5925], using ALPN to negotiate the use of AO. [[OPEN ISSUE: Is this something we want? Maybe in a separate specification.]]

## 6. Implementation Options

There are two primary implementation options for TCP-TLS:

- Implement all of TCP-TLS in the operating system kernel.

- Implement just the TCP-TLS negotiation option in the operating system kernel with an interface to tell the application that TCP-TLS has been negotiated and therefore that the application must negotiate TLS.

The former option obviously achieves easier deployment for applications, which don't have to do anything, but is more effort for kernel developers and requires a wider interface to the kernel to configure the TLS stack. The latter option is inherently more flexible but does not provide as immediate transparent deployment. It is also possible for systems to offer both options.

## 7. NAT/Firewall considerations

If use of TLS is negotiated, the data sent over TCP simply is TLS data in compliance with {{RFC5246}. Thus it is extremely likely to pass through NATs, firewalls, etc. The only kind of middlebox that is likely to cause a problem is one which does protocol enforcement that blocks TLS on arbitrary (non-443) ports but _also_ passes unknown TCP options. Although no doubt such devices do exist, because this is a common scenario, a client machine should be able to probe to determine if it is behind such a device relatively readily.

## 8. IANA Considerations

IANA [shall register/has registered] the TCP-ENO suboption XX for TCP-TLS.

IANA [shall register/has registered] the ALPN code point "tcpao" to indicate the use of TCP-TLS with TCP-AO.

## 9. Security Considerations

The mechanisms in this document are inherently vulnerable to active attack because an attacker can remove the TCP-TLS option, thus downgrading you to ordinary TCP.  Even when TCP-AO is used, all that is being provided is continuity of authentication from the initial handshake.  If some sort of external authentication mechanism was provided or certificates are used, then you might get some protection against active attack.

Once the TCP-TLS option has been negotiated, then the connection is resistant to active data injection attacks.  If TCP-AO is not used, then injected packets appear as bogus data at the TLS layer and will result in MAC errors followed by a fatal alert.  The result is that while data integrity is provided, the connection is not resistant to DoS attacks intended to terminate it.

If TCP-AO is used, then any bogus packets injected by an attacker will be rejected by the TCP-AO integrity check and therefore will never reach the TLS layer.  Thus, in this case, the connection is also resistant to DoS attacks, provided that endpoints require integrity protection for RST packets.  If endpoints accept unauthenticated RST, then no DoS protection is provided.

## 10. References

### 10.1. Normative References

[I-D.bittau-tcpinc-tcpeno]
Bittau, A., Boneh, D., Giffin, D., Handley, M., Mazieres, D., and E. Smith, "TCP-ENO: Encryption Negotiation Option", draft-bittau-tcpinc-tcpeno-02 (work in progress), September 2015.

[I-D.ietf-tls-applayerprotoneg]
Friedl, S., Popov, A., Langley, A., and S. Emile, "Transport Layer Security (TLS) Application Layer Protocol Negotiation Extension", draft-ietf-tls-applayerprotoneg-05 (work in progress), March 2014.

[I-D.ietf-tls-chacha20-poly1305]
Langley, A., Chang, W., Mavrogiannopoulos, N., Strombergson, J., and S. Josefsson, "The ChaCha20-Poly1305 AEAD Cipher for Transport Layer Security", draft-ietf-tls-chacha20-poly1305-00 (work in progress), June 2015.

[I-D.ietf-tls-session-hash]
Bhargavan, K., Delignat-Lavaud, A., Pironti, A., Langley, A., and M. Ray, "Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension", draft-ietf-tls-session-hash-06 (work in progress), July 2015.

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-08 (work in progress), August 2015.

[I-D.irtf-cfrg-curves]
Langley, A. and M. Hamburg, "Elliptic Curves for Security", draft-irtf-cfrg-curves-09 (work in progress), September 2015.

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

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <http://www.rfc-editor.org/info/rfc5116>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://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, <http://www.rfc-editor.org/info/rfc5705>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <http://www.rfc-editor.org/info/rfc5869>.

[RFC5925] Touch, J., Mankin, A., and R. Bonica, "The TCP Authentication Option", RFC 5925, DOI 10.17487/RFC5925, June 2010, <http://www.rfc-editor.org/info/rfc5925>.

[RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <http://www.rfc-editor.org/info/rfc7250>.

### 10.2. Informative References

[I-D.bittau-tcp-crypt]
Bittau, A., Boneh, D., Hamburg, M., Handley, M., Mazieres, D., and Q. Slack, "Cryptographic protection of TCP Streams (tcpcrypt)", draft-bittau-tcp-crypt-04 (work in progress), February 2014.

[I-D.ietf-tls-falsestart]
Langley, A., Modadugu, N., and B. Moeller, "Transport Layer Security (TLS) False Start", draft-ietf-tls-falsestart-00 (work in progress), May 2015.

[RFC5929] Altman, J., Williams, N., and L. Zhu, "Channel Bindings for TLS", RFC 5929, DOI 10.17487/RFC5929, July 2010, <http://www.rfc-editor.org/info/rfc5929>.

[RFC6919] Barnes, R., Kent, S., and E. Rescorla, "Further Key Words for Use in RFCs to Indicate Requirement Levels", RFC 6919, DOI 10.17487/RFC6919, April 2013, <http://www.rfc-editor.org/info/rfc6919>.

Author's Address

Eric Rescorla
Mozilla

EMail: ekr@rtfm.com