Network Working Group
Internet-Draft
Intended status: Informational
Expires: February 11, 2016

A. Bittau
D. Boneh
D. Giffin
Stanford University
M. Handley
University College London
D. Mazieres
Stanford University
E. Smith
Kestrel Institute

August 10, 2015

Interface Extensions for TCPINC
draft-bittau-tcpinc-api-00

Abstract

TCP-ENO negotiates encryption at the transport layer. It also defines a few parameters that are intended to be used or configured by applications. This document specifies operating system interfaces for access for these TCP-ENO parameters. We describe the interfaces in terms of socket options, the de facto standard API for adjusting per-connection behavior in TCP/IP, and sysctl, a popular mechanism for setting global defaults. Operating systems that lack socket or sysctl functionality can implement similar interfaces in their native frameworks, but should ideally adapt their interfaces from those presented in this document.

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 February 11, 2016.

Table of Contents

1. Introduction

The TCP Encryption Negotiation Option (TCP-ENO) [I-D.bittau-tcpinc-tcpeno] permits hosts to negotiate encryption of a TCP connection. One of TCP-ENO's use cases is to encrypt traffic transparently, unbeknownst to legacy applications. Transparent encryption requires no changes to existing APIs. However, other use cases require applications to interact with TCP-ENO. In particular:

- Transparent encryption protects only against passive eavesdroppers. Stronger security requires applications to authenticate a _Session ID_ value associated with each encrypted connection.

- Applications that have been updated to authenticate Session IDs must somehow advertise this fact to peers in a backward-compatible way. TCP-ENO carries a two-bit "application-aware" status for

this purpose, but this status is not accessible through existing interfaces.

- Applications employing TCP's simultaneous open feature need a way to supply a symmetry-breaking "tie-breaker" bit to TCP-ENO.

- System administrators and applications may wish to set and examine negotiation preferences, such as which encryption schemes (and perhaps versions) to enable and disable.

- Applications that perform their own encryption may wish to disable TCP-ENO entirely.

The remainder of this document describes an API through which systems can meet the above needs. The API extensions relate back to quantities defined by TCP-ENO.

2. API extensions

Application should access TCP-ENO options through the same mechanism they use to access other TCP configuration options, such as "TCP_NODELAY" [RFC0896]. With the popular sockets API, this mechanism consists of two socket options, "getsockopt" and "setsockopt", shown in Figure 1. Socket-based TCP-ENO implementations should define a set of new "option_name" values accessible at "level" "IPPROTO_TCP" (generally defined as 6, to match the IP protocol field).

```
int getsockopt(int socket, int level, int option_name,
               void *option_value, socklen_t *option_len);

int setsockopt(int socket, int level, int option_name,
               const void *option_value, socklen_t option_len);
```

Figure 1: Socket option API

Table 1 summarizes the new "option_name" arguments that TCP-ENO introduces to the socket option (or equivalent) system calls. For each option, the table lists whether it is read-only (R) or read-write (RW), as well as the type of the option's value. Read-write options, when read, always return the previously successfully written value or the default if they have not been written. Options of type "bytes" consist of a variable-length array of bytes, while options of type "int" consist of a small integer with the exact range indicated in parentheses. We discuss each option in more detail below.

Option name	RW	Type
TCPENO_ENABLED	RW	int (-1 - 1)
TCPENO_SESSID	R	bytes
TCPENO_NEGSPEC	R	int (32 - 255)
TCPENO_SPECS	RW	bytes
TCPENO_SELF_AWARE	RW	int (0 - 3)
TCPENO_PEER_AWARE	R	int (0 - 3)
TCPENO_TIEBREAKER	RW	int (0 - 1)
TCPENO_ROLE	R	int (0 - 1)
TCPENO_RAW	RW	bytes
TCPENO_TRANSCRIPT	R	bytes

Table 1: Suggested new IPPROTO_TCP socket options

TCPENO_ENABLED When set to 0, completely disables TCP-ENO regardless of any other socket option settings except "TCPENO_RAW". When set to 1, enables TCP-ENO. When set to -1 (which should be the default), uses a system default value to determine whether or not to enable TCP-ENO. This option must return an error after a SYN segment has already been sent.

TCPENO_SESSID Returns the session ID of the connection, as defined by the encryption spec in use. This option must return an error if encryption is disabled, the connection is not yet established, or the transport layer does not implement the negotiated spec.

TCPENO_NEGSPEC Returns the negotiated encryption spec for the current connection. As defined by TCP-ENO, the negotiated spec is the first valid suboption in the "B" host's SYN segment (without any suboption data for variable-length suboptions). This option must return an error if encryption is disabled or the connection is not yet established.

TCPENO_SPECS Allows the application to specify an ordered list of encryption specs different from the system default list. If the list is empty, TCP-ENO is disabled for the connection. Each byte in the list specifies one suboption type from 0x20-0xff. The list contains no suboption data for variable-length suboptions, only the one-byte spec identifier. The order of the list matters only for the host playing the "B" role. Implementations must return an error if an application attempts to set this option after the SYN segment has been sent. Implementations should return an error if any of the bytes are below 0x20 or are not implemented by the TCP stack.

TCPENO_SELF_AWARE The value is an integer from 0-3, allowing applications to specify the "aa" bits in the general suboption sent by the host. When listening on a socket, the value of this option applies to each accepted connection. Implementations must return an error if an application attempts to set this option after a SYN segment has been sent.

TCPENO_PEER_AWARE The value is an integer from 0-3 reporting the "aa" bits in the general suboption of the peer's segment. Implementations must return an error if an application attempts to read this value before the connection is established.

TCPENO_TIEBREAKER The value is a bit (0 or 1), indicating the value of the "b" bit to set in the host's general suboption. The "b" bit breaks the symmetry of simultaneous open to assign a unique role "A" or "B" to each end of the connection. The host that sets the "b" bit assumes the "B" role (which in non-simultaneous open is assigned to the passive opener). Implementations must return an error for this options after the SYN segment has already been sent.

TCPENO_ROLE The value is a bit (0 or 1). TCP-ENO defines two roles, "A" and "B", for the two ends of a connection. After a normal three-way handshake, the active opener is "A" and the passive opener is "B". Simultaneous open uses the tie-breaker bit to assign unique roles. This option returns 0 when the local host has the "A" role, and 1 when the local host has the "B" role. This call must return an error before the connection is established or if TCP-ENO has failed.

TCPENO_RAW This option is for use by library-level implementations of encryption specs. It allows applications to make use of the TCP-ENO option, potentially including encryption specs not supported by the transport layer, and then entirely bypass any TCP-level encryption so as to encrypt above the transport layer. The default value of this option is a 0-byte vector, which disables RAW mode. If the option is set to any other value, it disables all other socket options described in this section except for TCPENO_TRANSCRIPT.

The value of the option is a raw ENO option contents (without the kind and length) to be included in the host's SYN segment. In raw mode, the TCP layer considers negotiation successful when the two SYN segments both contain a suboption with the same encryption spec value "cs" >= 0x20. For an active opener in raw mode, the TCP layer automatically sends a two-byte minimal ENO option when negotiation is successful. Note that raw mode performs no sanity

checking on the "v" bits or any suboption data, and hence provides slightly less flexibility than a true TCP-level implementation.

TCPENO_TRANSCRIPT Returns the negotiation transcript as specified by TCP-ENO. Implementations must return an error if negotiation failed or has not yet completed.

In addition to these per-socket options, implementations should use "sysctl" or an equivalent mechanism to allow administrators to configure system-wide defaults for "TCPENO_ENABLED" and "TCPENO_SPECS". These parameters should be named "eno_enabled" and "eno_specs" and placed alongside most TCP parameters. For example, on BSD derived systems a suitable name would be "net.inet.tcp.eno_enabled" and "net.inet.tcp.eno_specs", while on Linux more appropriate names would be "net.ipv4.tcp_eno_enabled" and "net.ipv4.tcp_eno_specs".

Because initial deployment may run into issues with middleboxes or incur slowdown for unnecessary double-encryption, implementations should also allow ENO to be blacklisted for particular local and remote ports, via sysctl on "net.inet.tcp.eno_bad_localport" and "net.inet.tcp.eno_bad_remoteport" (or the equivalent under "net.ipv4" for linux), both of which consist of a list of TCP port numbers on which to disable TCP-ENO by default. For example the following command:

```
sysctl net.inet.tcp.eno_bad_remoteport=443,993
```

would disable ENO encryption on outgoing connections to ports 443 and 993 (which use application-layer encryption for TLS and IMAP, respectively).

The per-socket "TCPENO_ENABLED" option, if not -1, should override both the "eno_enabled" and port-range sysctls.

3. Automatic configuration protocol

TCP-ENO is designed to fail by reverting to unencrypted TCP. Such behavior is necessary for incremental deployment, and is no worse than the status quo in which there is no TCP-layer encryption. However, one outcome worse than the status quo would be to for TCP-ENO connections to fail completely where unenecrypted connections would work. Fortunately, if TCP-ENO is not supported by both endpoints, or if middleboxes strip the ENO option from packets, then implementations simply revert to unencrypted TCP upon receiving a SYN or initial ACK segment without an ENO option. This fallback approach also applies to interception proxies [RFC3040], which typically

terminate TCP connections and hence will not include ENO in their SYN segments if they do not know about it.

However, given that the goal of TCP-ENO is to encrypt previously plaintext traffic, there is always the possibility that a middlebox performing deep packet inspection could shut down a connection because the ciphertext does not resemble an expected higher-level application protocol such as HTTP. Such middleboxes would cause TCP-ENO connections to fail. Systems may wish to probe the network so as to enable TCP-ENO only in places where middleboxes do not induce such failures.

A precedent for probing middlebox behavior is the STUN protocol [RFC5389], which applications use to characterize NAT. STUN relies on having a known, publicly-accessible server beyond any locally administered middleboxes. STUN is typically invoked by applications that require peer-to-peer communication to decide whether they can accept incoming connections. For TCP-ENO, which affects all TCP connections, it makes more sense to probe for network compatibility at the time network interfaces are configured by DHCP [RFC2131], stateless address autoconfiguration [RFC4862], or other mechanisms. Many DHCP implementation already provide hooks through which such probes can be configured.

Like STUN, TCP-ENO probing requires a known external server running an agreed upon protocol. We suggests using HTTP as the protocol, and responding to the path "/tcp-eno/session-id" with a response of type "text/plain". Upon successful TCP-ENO negotiation, servers should reply with the string "encrypted " followed by a lower-case hexadecimal encoding of the tcpcrypt session ID followed by a newline. For connection on which TCP-ENO fails, the same path should return the string "unencrypted\n" (with no session ID). If such a request works with TCP-ENO disabled but hangs or resets with TCP-ENO enabled, then TCP-ENO should be disabled for the host. Otherwise, if probes succeed, even if they return "unencrypted", TCP-ENO should be enabled (for the possible benefit of local connections), as middleboxes may simply be stripping off the option.

Hosts should perform the above probe twice, using both port 80 and a different port, we suggest 8080, on the same server. Given the prevalence of interception proxies on port 80, port 80 may experience entirely different failure modes from other ports. If the port 80 probe fails, TCP-ENO should be disabled for port 80. If the other probe fails, TCP-ENO should be disabled entirely.

4. Examples

This section provides examples of how applications might authenticate session IDs. Authentication requires exchanging messages over the TCP connection, and hence is not backwards compatible with existing application protocols. To fall back to opportunistic encryption in the event that both applications have not been updated to authenticate the session ID, TCP-ENO provides the application-aware bits. To signal it has been upgraded to support application-level authentication, applications should set "TCPENO_SELF_AWARE" to 1 before opening a connection. An application should then check that "TCPENO_PEER_AWARE" is non-zero before attempting to send authenticators that would otherwise be misinterpreted as application data.

4.1. Cookie-based authentication

In cookie-based authentication, a client and server both share a cryptographically strong random or pseudo-random secret known as a "cookie". Such a cookie is preferably at least 128 bits long. To authenticate a session ID using a cookie, each computes and sends the following value to the other side:

```
authenticator = PRF(cookie, role || session-ID)
```

Here "PRF" is a psueo-random function such as HMAC-SHA-256 [RFC6234]. "role" is the byte 0 or 1, as returned by the "TCPENO_ROLE" socket options. "session-ID" is the session ID returned by the "TCPENO_SESSID" session ID. The symbol "||" denotes concatenation. Each side must verify that the other side's authenticator is correct. Assuming the authenticators are correct, applications can rely on the TCP-layer encryption for resistance against active network attackers.

Note that if the same cookie is used in other contexts besides session ID authentication, appropriate domain separation should be employed, such as prefixing "role || session-ID" with a unique prefix to ensure "authenticator" cannot be used out of context.

4.2. Signature-based authentication

In signature-based authentication, one or both endpoints of a connection possess a private signature key the public half of which is known to or verifiable by the other endpoint. To authenticate itself, the host with a private key computes the following signature:

```
authenticator = Sign(PrivKey, role || session-ID)
```

The other end verifies this value using the corresponding public key. Whichever side validates an authenticator in this way knows that the other side belongs to a host that possesses the appropriate signature key.

Once again, if the same signature key is used in other contexts besides session ID authentication, appropriate domain separation should be employed, such as prefixing "role || session-ID" with a unique prefix to ensure "authenticator" cannot be used out of context.

5. Security considerations

The TCP-ENO specification discusses several important security considerations that this document incorporates by reference. The most important one, which bears reiterating, is that until and unless a session ID has been authenticated, TCP-ENO is vulnerable to an active network attacker, through either a downgrade or active man-in-the-middle attack.

Because of this vulnerability to active network attackers, it is critical that implementations return appropriate errors for socket options when TCP-ENO is not enabled. Equally critical is that applications must never use these socket options without checking for errors.

Applications with high security requirements that rely on TCP-ENO for security must either fail or fallback to application-layer encryption if TCP-ENO fails or session IDs authentication fails.

6. Acknowledgments

This work was funded by DARPA CRASH under contract #N66001-10-2-4088.

7. References

7.1. 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-01 (work in progress), August 2015.

7.2. Informative References

[RFC0896] Nagle, J., "Congestion Control in IP/TCP Internetworks", RFC 896, DOI 10.17487/RFC0896, January 1984, <http://www.rfc-editor.org/info/rfc896>.

[RFC2131] Droms, R., "Dynamic Host Configuration Protocol", RFC 2131, DOI 10.17487/RFC2131, March 1997, <http://www.rfc-editor.org/info/rfc2131>.

[RFC3040] Cooper, I., Melve, I., and G. Tomlinson, "Internet Web Replication and Caching Taxonomy", RFC 3040, DOI 10.17487/RFC3040, January 2001, <http://www.rfc-editor.org/info/rfc3040>.

[RFC4862] Thomson, S., Narten, T., and T. Jinmei, "IPv6 Stateless Address Autoconfiguration", RFC 4862, DOI 10.17487/RFC4862, September 2007, <http://www.rfc-editor.org/info/rfc4862>.

[RFC5389] Rosenberg, J., Mahy, R., Matthews, P., and D. Wing, "Session Traversal Utilities for NAT (STUN)", RFC 5389, DOI 10.17487/RFC5389, October 2008, <http://www.rfc-editor.org/info/rfc5389>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <http://www.rfc-editor.org/info/rfc6234>.

Authors' Addresses

Andrea Bittau
Stanford University
353 Serra Mall, Room 288
Stanford, CA 94305
US

Email: bittau@cs.stanford.edu

Dan Boneh
Stanford University
353 Serra Mall, Room 475
Stanford, CA 94305
US

Email: dabo@cs.stanford.edu

Daniel B. Giffin
Stanford University
353 Serra Mall, Room 288
Stanford, CA 94305
US

Email: dbg@scs.stanford.edu

Mark Handley
University College London
Gower St.
London WC1E 6BT
UK

Email: M.Handley@cs.ucl.ac.uk

David Mazieres
Stanford University
353 Serra Mall, Room 290
Stanford, CA 94305
US

Email: dm@uun.org

Eric W. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304
US

Email: eric.smith@kestrel.edu

Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 3, 2016

A. Bittau
D. Boneh
D. Giffin
Stanford University
M. Handley
University College London
D. Mazieres
Stanford University
E. Smith
Kestrel Institute

March 2, 2016

Interface Extensions for TCP-ENO
draft-bittau-tcpinc-api-01

Abstract

TCP-ENO negotiates encryption at the transport layer. It also defines a few parameters that are intended to be used or configured by applications. This document specifies operating system interfaces for access to these TCP-ENO parameters. We describe the interfaces in terms of socket options, the de facto standard API for adjusting per-connection behavior in TCP/IP, and sysctl, a popular mechanism for setting global defaults. Operating systems that lack socket or sysctl functionality can implement similar interfaces in their native frameworks, but should ideally adapt their interfaces from those presented in this document.

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 September 3, 2016.

Table of Contents

1. Introduction

The TCP Encryption Negotiation Option (TCP-ENO) [I-D.ietf-tcpinc-tcpeno] permits hosts to negotiate encryption of a TCP connection. One of TCP-ENO's use cases is to encrypt traffic transparently, unbeknownst to legacy applications. Transparent encryption requires no changes to existing APIs. However, other use cases require applications to interact with TCP-ENO. In particular:

- Transparent encryption protects only against passive eavesdroppers. Stronger security requires applications to authenticate a _Session ID_ value associated with each encrypted connection.

- Applications that have been updated to authenticate Session IDs must somehow advertise this fact to peers in a backward-compatible way. TCP-ENO carries a two-bit "application-aware" status for

this purpose, but this status is not accessible through existing interfaces.

- Applications employing TCP's simultaneous open feature need a way to supply a symmetry-breaking "role-override" bit to TCP-ENO.

- System administrators and applications may wish to set and examine negotiation preferences, such as which encryption schemes (and perhaps versions) to enable and disable.

- Applications that perform their own encryption may wish to disable TCP-ENO entirely.

The remainder of this document describes an API through which systems can meet the above needs. The API extensions relate back to quantities defined by TCP-ENO.

2. API extensions

This section describes an API for per-connection options, followed by a discussion of system-wide configuration options.

2.1. Per-connection options

Application should access TCP-ENO options through the same mechanism they use to access other TCP configuration options, such as "TCP_NODELAY" [RFC0896]. With the popular sockets API, this mechanism consists of two socket options, "getsockopt" and "setsockopt", shown in Figure 1. Socket-based TCP-ENO implementations should define a set of new "option_name" values accessible at "level" "IPPROTO_TCP" (generally defined as 6, to match the IP protocol field).

```
int getsockopt(int socket, int level, int option_name,
               void *option_value, socklen_t *option_len);

int setsockopt(int socket, int level, int option_name,
               const void *option_value, socklen_t option_len);
```

Figure 1: Socket option API

Table 1 summarizes the new "option_name" arguments that TCP-ENO introduces to the socket option (or equivalent) system calls. For each option, the table lists whether it is read-only (R) or read-write (RW), as well as the type of the option's value. Read-write options, when read, always return the previously successfully written value or the default if they have not been written. Options of type "bytes" consist of a variable-length array of bytes, while options of

type "int" consist of a small integer with the exact range indicated in parentheses. We discuss each option in more detail below.

Option name	RW	Type
TCP_ENO_ENABLED	RW	int (-1 - 1)
TCP_ENO_SESSID	R	bytes
TCP_ENO_NEGSPEC	R	int (32 - 127)
TCP_ENO_SPECS	RW	bytes
TCP_ENO_SELF_AWARE	RW	int (0 - 3)
TCP_ENO_PEER_AWARE	R	int (0 - 3)
TCP_ENO_ROLEOVERRIDE	RW	int (0 - 1)
TCP_ENO_ROLE	R	int (0 - 1)
TCP_ENO_LOCAL_NAME	R	bytes
TCP_ENO_PEER_NAME	R	bytes
TCP_ENO_RAW	RW	bytes
TCP_ENO_TRANSCRIPT	R	bytes

Table 1: Suggested new IPPROTO_TCP socket options

The socket options must return errors under certain circumstances. These errors are mapped to three suggested error codes shown in Table 2. Most socket-based systems will already have constants for these errors. Non-socket systems should use existing error codes corresponding to the same conditions. "EINVAL" is the existing error returned when setting options on a closed socket. "EISCONN" corresponds to calling connect a second time, while "ENOTCONN" corresponds to requesting the peer address of an unconnected socket.

Symbol	Description
EINVAL	General error signifying bad parameters
EISCONN	Option no longer valid because socket is connected
ENOTCONN	Option not (yet) valid because socket not connected

Table 2: Suggested error codes

TCP_ENO_ENABLED When set to 0, completely disables TCP-ENO regardless of any other socket option settings except "TCP_ENO_RAW". When set to 1, enables TCP-ENO. If set to -1, use a system-wide default determined at the time of an "accept" or "connect" system call, as described in Section 2.2. This option must return an error ("EISCONN") after a SYN segment has already been sent.

TCP_ENO_SESSID Returns the session ID of the connection, as defined by the encryption spec in use. This option must return an error if encryption is disabled ("EINVAL"), the connection is not yet established ("ENOTCONN"), or the transport layer does not implement the negotiated spec ("EINVAL").

TCP_ENO_NEGSPEC Returns the 7-bit code point of the negotiated encryption spec for the current connection. As defined by TCP-ENO, the negotiated spec is the last valid suboption in the "B" host's SYN segment. This option must return an error if encryption is disabled ("EINVAL") or the connection is not yet established ("ENOTCONN").

TCP_ENO_SPECS Allows the application to specify an ordered list of encryption specs different from the system default list. If the list is empty, TCP-ENO is disabled for the connection. Each byte in the list specifies one suboption type from 0x20-0xff. The list contains no suboption data for variable-length suboptions, only the one-byte spec identifier. The high bit ("v") in these bytes is ignored unless future implementations of encryption specs assign it special meaning. The order of the list matters only for the host playing the "B" role. Implementations must return an error ("EISCONN") if an application attempts to set this option after the SYN segment has been sent. Implementations should return an error ("EINVAL") if any of the bytes are below 0x20 or are not implemented by the TCP stack.

TCP_ENO_SELF_AWARE The value is an integer from 0-3, allowing applications to specify the "aa" bits in the general suboption sent by the host. When listening on a socket, the value of this option applies to each accepted connection. The default value should be 0. Implementations must return an error ("EISCONN") if an application attempts to set this option after a SYN segment has been sent.

TCP_ENO_PEER_AWARE The value is an integer from 0-3 reporting the "aa" bits in the general suboption of the peer's segment. Implementations must return an error ("ENOTCONN") if an application attempts to read this value before the connection is established.

TCP_ENO_ROLEOVERRIDE The value is a bit (0 or 1), indicating the value of the "b" bit to set in the host's general suboption. The "b" bit breaks the symmetry of simultaneous open to assign a unique role "A" or "B" to each end of the connection. The host that sets the "b" bit assumes the "B" role (which in non-simultaneous open is by default assigned to the passive opener). Implementations must return an error ("EISCONN") for attempts to

set this option after the SYN segment has already been sent. The default value should be 0.

TCP_ENO_ROLE The value is a bit (0 or 1). TCP-ENO defines two roles, "A" and "B", for the two ends of a connection. After a normal three-way handshake, the active opener is "A" and the passive opener is "B". Simultaneous open uses the role-override bit to assign unique roles. This option returns 0 when the local host has the "A" role, and 1 when the local host has the "B" role. This call must return an error before the connection is established ("ENOTCONN") or if TCP-ENO has failed ("EINVAL").

TCP_ENO_LOCAL_NAME Returns the concatenation of the TCP_ENO_ROLE byte and the TCP_ENO_SESSID. This provides a unique name for the local end of the connection.

TCP_ENO_PEER_NAME Returns the concatenation of the negation of the TCP_ENO_ROLE byte and the TCP_ENO_SESSID. This is the same value as returned by TCP_ENO_LOCAL_NAME on the other host, and hence provides a unique name for the remote end of the connection.

TCP_ENO_RAW This option is for use by library-level implementations of encryption specs. It allows applications to make use of the TCP-ENO option, potentially including encryption specs not supported by the transport layer, and then entirely bypass any TCP-level encryption so as to encrypt above the transport layer. The default value of this option is a 0-byte vector, which disables RAW mode. If the option is set to any other value, it disables all other socket options described in this section except for TCP_ENO_TRANSCRIPT.

The value of the option is a raw ENO option contents (without the kind and length) to be included in the host’s SYN segment. In raw mode, the TCP layer considers negotiation successful when the two SYN segments both contain a suboption with the same encryption spec value "cs" >= 0x20. For an active opener in raw mode, the TCP layer automatically sends a two-byte minimal ENO option when negotiation is successful. Note that raw mode performs no sanity checking on the "v" bits or any suboption data, and hence provides slightly less flexibility than a true TCP-level implementation.

TCP_ENO_TRANSCRIPT Returns the negotiation transcript as specified by TCP-ENO. Implementations must return an error if negotiation failed ("EINVAL") or has not yet completed ("ENOTCONN").

2.2. System-wide options

In addition to these per-socket options, implementations should use "sysctl" or an equivalent mechanism to allow administrators to configure a default value for "TCP_ENO_SPECS", as well as default behavior for when "TCP_ENO_ENABLED" is -1. Table 3 provides a table of suggested parameters. The type "words" corresponds to a list of 16-bit unsigned words representing TCP port numbers (similar to the "baddynamic" sysctls that, on some operating systems, blacklist automatic assignment of particular ports). These parameters should be placed alongside most TCP parameters. For example, on BSD derived systems a suitable name would be "net.inet.tcp.eno_specs", while on Linux a more appropriate name would be "net.ipv4.tcp_eno_specs".

Name	Type
eno_specs	bytes
eno_enable_connect	int (0 - 1)
eno_enable_listen	int (0 - 1)
eno_bad_connect_ports	words
eno_bad_listen_ports	words

Table 3: Suggested sysctl values

"eno_specs" is simply a string of bytes, and provides the default value for the "TCP_ENO_SPECS" socket option. If "TCP_ENO_SPECS" is non-empty, the remaining sysctls determine whether to attempt TCP-ENO negotiation when the "TCP_ENO_ENABLED" option is -1 (the default), using the following rules.

- On active openers: If "eno_enable_connect" is 0, then TCP-ENO is disabled. If the remote port number is in "eno_bad_connect_ports", then TCP-ENO is disabled. Otherwise, the host attempts to use TCP-ENO.

- On passive openers: If "eno_enable_listen" is 0, then TCP-ENO is disabled. Otherwise, if the local port is in "eno_bad_listen_ports", then TCP-ENO is disabled. Otherwise, if the host receives an SYN segment with an ENO option containing compatible encryption specs, it attempts negotiation.

Because initial deployment may run into issues with middleboxes or incur slowdown for unnecessary double-encryption, sites may wish to blacklist particular ports. For example the following command:

```
sysctl net.inet.tcp.eno_bad_connect_ports=443,993
```

would disable ENO encryption on outgoing connections to ports 443 and 993 (which use application-layer encryption for HTTP and IMAP, respectively). If the per-socket "TCP_ENO_ENABLED" is not -1, it overrides the sysctl values.

On a server, running:

```
sysctl net.inet.tcp.eno_bad_listen_ports=443
```

makes it possible to disable TCP-ENO for incoming HTTPS connection without modifying the web server to set "TCP_ENO_ENABLED" to 0.

3. Examples

This section provides examples of how applications might authenticate session IDs. Authentication requires exchanging messages over the TCP connection, and hence is not backwards compatible with existing application protocols. To fall back to opportunistic encryption in the event that both applications have not been updated to authenticate the session ID, TCP-ENO provides the application-aware bits. To signal it has been upgraded to support application-level authentication, an application should set "TCP_ENO_SELF_AWARE" to 1 before opening a connection. An application should then check that "TCP_ENO_PEER_AWARE" is non-zero before attempting to send authenticators that would otherwise be misinterpreted as application data.

3.1. Cookie-based authentication

In cookie-based authentication, a client and server both share a cryptographically strong random or pseudo-random secret known as a "cookie". Such a cookie is preferably at least 128 bits long. To authenticate a session ID using a cookie, each host computes and sends the following value to the other side:

```
authenticator = PRF(cookie, local-name)
```

Here "PRF" is a pseudo-random function such as HMAC-SHA-256 [RFC6234]. "local-name" is the result of the "TCP_ENO_LOCAL_NAME" socket option. Each side must verify that the other side’s authenticator is correct. To do so, software obtains the remote host’s local name via the "TCP_ENO_PEER_NAME" socket option. Assuming the authenticators are correct, applications can rely on the TCP-layer encryption for resistance against active network attackers.

Note that if the same cookie is used in other contexts besides session ID authentication, appropriate domain separation must be

employed, such as prefixing "local-name" with a unique prefix to ensure "authenticator" cannot be used out of context.

3.2. Signature-based authentication

In signature-based authentication, one or both endpoints of a connection possess a private signature key the public half of which is known to or verifiable by the other endpoint. To authenticate itself, the host with a private key computes the following signature:

```
authenticator = Sign(PrivKey, local-name)
```

The other end verifies this value using the corresponding public key. Whichever side validates an authenticator in this way knows that the other side belongs to a host that possesses the appropriate signature key.

Once again, if the same signature key is used in other contexts besides session ID authentication, appropriate domain separation should be employed, such as prefixing "local-name" with a unique prefix to ensure "authenticator" cannot be used out of context.

4. Security considerations

The TCP-ENO specification discusses several important security considerations that this document incorporates by reference. The most important one, which bears reiterating, is that until and unless a session ID has been authenticated, TCP-ENO is vulnerable to an active network attacker, through either a downgrade or active man-in-the-middle attack.

Because of this vulnerability to active network attackers, it is critical that implementations return appropriate errors for socket options when TCP-ENO is not enabled. Equally critical is that applications must never use these socket options without checking for errors.

Applications with high security requirements that rely on TCP-ENO for security must either fail or fall back to application-layer encryption if TCP-ENO fails or session IDs authentication fails.

5. Acknowledgments

This work was funded by DARPA CRASH under contract #N66001-10-2-4088.

6. References

6.1. Normative References

[I-D.ietf-tcpinc-tcpeno]
Bittau, A., Boneh, D., Giffin, D., Handley, M., Mazieres, D., and E. Smith, "TCP-ENO: Encryption Negotiation Option", draft-ietf-tcpinc-tcpeno-01 (work in progress), February 2016.

6.2. Informative References

[RFC0896] Nagle, J., "Congestion Control in IP/TCP Internetworks", RFC 896, DOI 10.17487/RFC0896, January 1984, <http://www.rfc-editor.org/info/rfc896>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <http://www.rfc-editor.org/info/rfc6234>.

Authors' Addresses

Andrea Bittau
Stanford University
353 Serra Mall, Room 288
Stanford, CA 94305
US

Email: bittau@cs.stanford.edu

Dan Boneh
Stanford University
353 Serra Mall, Room 475
Stanford, CA 94305
US

Email: dabo@cs.stanford.edu

Daniel B. Giffin
Stanford University
353 Serra Mall, Room 288
Stanford, CA 94305
US

Email: dbg@scs.stanford.edu

Mark Handley
University College London
Gower St.
London WC1E 6BT
UK

Email: M.Handley@cs.ucl.ac.uk

David Mazieres
Stanford University
353 Serra Mall, Room 290
Stanford, CA 94305
US

Email: dm@uun.org

Eric W. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304
US

Email: eric.smith@kestrel.edu

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 6, 2016

A. Bittau
D. Boneh
D. Giffin
M. Hamburg
Stanford University
M. Handley
University College London
D. Mazieres
Q. Slack
Stanford University
E. Smith
Kestrel Institute

November 3, 2015

Cryptographic protection of TCP Streams (tcpcrypt)
draft-ietf-tcpinc-tcpcrypt-00

Abstract

This document specifies tcpcrypt, a cryptographic protocol that protects TCP payload data and is negotiated by means of the TCP Encryption Negotiation Option (TCP-ENO) [I-D.ietf-tcpinc-tcpeno]. Tcpcrypt coexists with middleboxes by tolerating resegmentation, NATs, and other manipulations of the TCP header. The protocol is self-contained and specifically tailored to TCP implementations, which often reside in kernels or other environments in which large external software dependencies can be undesirable. Because of option size restrictions, the protocol requires one additional one-way message latency to perform key exchange. However, this cost is avoided between two hosts that have recently established a previous tcpcrypt connection.

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 May 6, 2016.

Table of Contents

1. Requirements language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Introduction

This document describes tcpcrypt, an extension to TCP for cryptographic protection of session data. Tcpcrypt was designed to meet the following goals:

- Meet the requirements of the TCP Encryption Negotiation Option (TCP-ENO) [I-D.ietf-tcpinc-tcpeno] for protecting connection data.
- Be amenable to small, self-contained implementations inside TCP stacks.
- Avoid unnecessary round trips.
- As much as possible, prevent connection failure in the presence of NATs and other middleboxes that might normalize traffic or otherwise manipulate TCP segments.
- Operate independently of IP addresses, making it possible to authenticate resumed TCP connections even when either end changes IP address.

3. Encryption protocol

This section describes the tcpcrypt protocol at an abstract level, so as to provide an overview and facilitate analysis. The next section specifies the byte formats of all messages.

3.1. Cryptographic algorithms

Setting up a tcpcrypt connection employs three types of cryptographic algorithms:

- A _key agreement scheme_ is used with a short-lived public key to agree upon a shared secret.

- An _extract function_ is used to generate a pseudo-random key from some initial keying material, typically the output of the key agreement scheme. The notation Extract(S, IKM) denotes the output of the extract function with salt S and initial keying material IKM.

- A _collision-resistant pseudo-random function (CPRF)_ is used to generate multiple cryptographic keys from a pseudo-random key, typically the output of the extract function. We use the notation CPRF(K, CONST, L) to designate the output of L bytes of the pseudo-random function identified by key K on CONST. A collision-resistant function is one on which, for sufficiently large L, an attacker cannot find two distinct inputs K_1, CONST_1 and K_2, CONST_2 such that CPRF(K_1, CONST_1, L) = CPRF(K_2, CONST_2, L). Collision resistance is important to assure the uniqueness of Session IDs, which are generated using the CPRF.

The Extract and CPRF functions used by default are the Extract and Expand functions of HKDF [RFC5869]. These are defined as follows in terms of the PRF "HMAC-Hash(key, value)" for a negotiated "Hash" function:

```
HKDF-Extract(salt, IKM) -> PRK
    PRK = HMAC-Hash(salt, IKM)

HKDF-Expand(PRK, CONST, L) -> OKM
   T(0) = empty string (zero length)
   T(1) = HMAC-Hash(PRK, T(0) | CONST | 0x01)
   T(2) = HMAC-Hash(PRK, T(1) | CONST | 0x02)
   T(3) = HMAC-Hash(PRK, T(2) | CONST | 0x03)
   ...

   OKM  = first L octets of T(1) | T(2) | T(3) | ...
```

Figure 1: The symbol | denotes concatenation, and the counter concatenated with CONST is a single octet.

Once tcpcrypt has been successfully set up, we say the connection moves to an ENCRYPTING phase, where it employs an _authenticated

encryption mode_ to encrypt and integrity-protect all application data.

Note that public-key generation, public-key encryption, and shared-secret generation all require randomness. Other tcpcrypt functions may also require randomness, depending on the algorithms and modes of operation selected. A weak pseudo-random generator at either host will compromise tcpcrypt's security. Thus, any host implementing tcpcrypt MUST have a cryptographically-secure source of randomness or pseudo-randomness.

3.2. Roles

Tcpcrypt transforms a single pseudo-random key (PRK) into cryptographic session keys for each direction. Doing so requires an asymmetry in the protocol, as the key derivation function must be perturbed differently to generate different keys in each direction. Tcpcrypt includes other asymmetries in the roles of the two hosts, such as the process of negotiating algorithms (e.g., proposing vs. selecting cipher suites).

To establish roles for the hosts, tcpcrypt depends on TCP-ENO [I-D.ietf-tcpinc-tcpeno]. As part of the negotiation process, TCP-ENO assigns hosts unique roles abstractly called "A" at one end of the connection and "B" at the other. Generally, an active opener plays the "A" role and a passive opener plays the "B" role, though an additional mechanism breaks the symmetry of simultaneous open. This document adopts the terms "A" and "B" to identify each end of a connection uniquely, following TCP-ENO's designation.

3.3. Protocol negotiation

Tcpcrypt also depends on TCP-ENO [I-D.ietf-tcpinc-tcpeno] to negotiate the use of tcpcrypt and a particular key agreement scheme. TCP-ENO negotiates an _encryption spec_ by means of suboptions embedded in SYN segments. Each suboption is identified by a byte consisting of a seven-bit _encryption spec identifier_ value, "cs", and a one-bit additional data indicator, "v". This document reserves and associates four "cs" values with tcpcrypt, as listed in Table 1; future standards can associate additional values with tcpcrypt.

A TCP connection MUST employ tcpcrypt and transition to the ENCRYPTING phase when and only when:

1. The TCP-ENO negotiated spec contains a "cs" value associated with tcpcrypt, and

2. The presence of variable-length data matches the suboption usage.

Specifically, when the "cs" value is "TCPCRYPT_RESUME", whose use is described in Section 3.5, there MUST be associated data (i.e., "v" MUST be 1). For all other "cs" values specified in this document, there MUST NOT be additional suboption data (i.e., "v" MUST be 0). Future "cs" values associated with tcpcrypt might or might not specify the use of associated data. Tcpcrypt implementations MUST ignore suboptions whose "cs" and "v" values do not agree as specified in this paragraph.

In normal usage, an active opener that wishes to negotiate the use of tcpcrypt will include an ENO option in its SYN segment; that option will include the tcpcrypt suboptions corresponding to the key-agreement schemes it is willing to enable, and possibly also a resumption suboption. The active opener MAY additionally include suboptions indicating support for encryption protocols other than tcpcrypt, as well as other general options as specified by TCP-ENO.

If a passive opener receives an ENO option including tcpcrypt suboptions it supports, it MAY then attach an ENO option to its SYN-ACK segment, including _solely_ the suboption it wishes to enable.

Once two hosts have exchanged SYN segments, the _negotiated spec_ is the last spec identifier in the SYN segment of host B (that is, the passive opener in the absence of simultaneous open) that also occurs in that of host A. If there is no such spec, hosts MUST disable TCP-ENO and tcpcrypt.

3.4. Key exchange

Following successful negotiation of a tcpcrypt spec, all further signaling is performed in the Data portion of TCP segments. If the negotiated spec is not TCPCRYPT_RESUME, the two hosts perform key exchange through two messages, INIT1 and INIT2, at the start of host A's and host B's data streams, respectively. INIT1 or INIT2 can span multiple TCP segments and need not end at a segment boundary. However, the segment containing the last byte of an INIT1 or INIT2 message SHOULD have TCP's PSH bit set.

The key exchange protocol, in abstract, proceeds as follows:

```
A -> B:  init1 = { INIT1_MAGIC, sym-cipher-list, N_A, PK_A }
B -> A:  init2 = { INIT2_MAGIC, sym-cipher, N_B, PK_B }
```

The format of these messages is specified in detail in Section 4.1.

The parameters are defined as follows:

- o sym-cipher-list: a list of symmetric ciphers (AEAD algorithms) acceptable to host A. These are specified in Table 2.

- o sym-cipher: the symmetric cipher selected by B from the sym-cipher-list sent by A.

- o N_A, N_B: nonces chosen at random by A and B, respectively.

- o PK_A, PK_B: ephemeral public keys for A and B, respectively. These, as well as their corresponding private keys, are short-lived values that SHOULD be refreshed periodically and SHOULD NOT ever be written to persistent storage.

The pre-master secret (PMS) is defined to be the result of the key-agreement algorithm whose inputs are the local host's ephemeral private key and the remote host's ephemeral public key. For example, host A would compute PMS using its own private key (not transmitted) and host B's public key, PK_B.

The two sides then compute a pseudo-random key (PRK), from which all session keys are derived, as follows:

```
param  := { eno-transcript, init1, init2 }
PRK    := Extract (N_A, { param, PMS })
```

Above, "eno-transcript" is the protocol-negotiation transcript defined in TCP-ENO; "init1" and "init2" are the transmitted encodings of the INIT1 and INIT2 messages described in Section 4.1.

A series of "session secrets" and corresponding Session IDs are then computed as follows:

```
ss[0] := PRK
ss[i] := CPRF (ss[i-1], CONST_NEXTK, K_LEN)

SID[i] := CPRF (ss[i], CONST_SESSID, K_LEN)
```

The value ss[0] is used to generate all key material for the current connection. SID[0] is the Session ID for the current connection, and will with overwhelming probability be unique for each individual TCP connection. The most computationally expensive part of the key exchange protocol is the public key cipher. The values of ss[i] for i > 0 can be used to avoid public key cryptography when establishing subsequent connections between the same two hosts, as described in Section 3.5. The CONST values are constants defined in Table 3. The K_LEN values depend on the tcpcrypt spec in use, and are specified in Figure 3.

Given a session secret, ss, the two sides compute a series of master keys as follows:

```
mk[0] := CPRF (ss, CONST_REKEY, K_LEN)
mk[i] := CPRF (mk[i-1], CONST_REKEY, K_LEN)
```

Finally, each master key mk is used to generate keys for authenticated encryption for the "A" and "B" roles. Key k_ab is used by host A to encrypt and host B to decrypt, while k_ba is used by host B to encrypt and host A to decrypt.

```
k_ab := CPRF(mk, CONST_KEY_A, ae_keylen)
k_ba := CPRF(mk, CONST_KEY_B, ae_keylen)
```

The ae_keylen value depends on the authenticated-encryption algorithm selected, and is given under "Key Length" in Table 2.

HKDF is not used directly for key derivation because tcpcrypt requires multiple expand steps with different keys. This is needed for forward secrecy, so that ss[n] can be forgotten once a session is established, and mk[n] can be forgotten once a session is rekeyed.

There is no "key confirmation" step in tcpcrypt. This is not required because tcpcrypt's threat model includes the possibility of a connection to an adversary. If key negotiation is compromised and yields two different keys, all subsequent frames will be ignored due failed integrity checks, causing the application's connection to hang. This is not a new threat because in plain TCP, an active attacker could have modified sequence and acknowledgement numbers to hang the connection anyway.

3.5. Session caching

When two hosts have already negotiated session secret ss[i-1], they can establish a new connection without public-key operations using ss[i]. A host wishing to request this facility will include in its SYN segment an ENO option whose last suboption contains the spec identifier TCPCRYPT_RESUME:

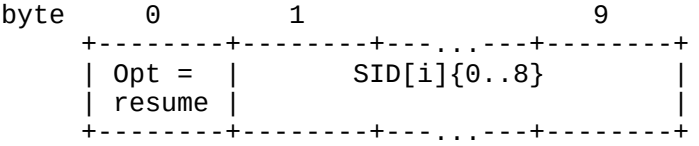


Figure 2: ENO suboption used to initiate session resumption

Above, the "resume" value is the byte whose lower 7 bits are TCPCRYPT_RESUME and whose top bit "v" is 1 (indicating variable-length data follows). The remainder of the suboption is filled with the first nine bytes of the Session ID SID[i].

A host SHOULD also include ENO suboptions describing the key-agreement schemes it supports in addition to a resume suboption, so as to fall back to full key exchange in the event that session resumption fails.

Which symmetric keys a host uses for transmitted segments is determined by its role in the original session ss[0]. It does not depend on the role it plays in the current session. For example, if a host had the "A" role in the first session, then it uses k_ab for sending segments and k_ba for receiving.

After using ss[i] to compute mk[0], implementations SHOULD compute and cache ss[i+1] for possible use by a later session, then erase ss[i] from memory. Hosts SHOULD keep ss[i+1] around for a period of time until it is used or the memory needs to be reclaimed. Hosts SHOULD NOT write a cached ss[i+1] value to non-volatile storage.

It is an implementation-specific issue as to how long ss[i+1] should be retained if it is unused. If the passive opener evicts it from cache before the active opener does, the only cost is the additional ten bytes to send the resumption suboption in the next connection. The behavior then falls back to a normal public-key handshake.

The active opener MUST use the lowest value of "i" that has not already appeared in a resumption suboption exchanged with the same host and for the same pre-session seed.

If the passive opener recognizes SID[i] and knows ss[i], it SHOULD respond with an ENO option containing a dataless resumption suboption; that is, the suboption whose "cs" value is TCPCRYPT_RESUME and whose "v" bit is zero.

If the passive opener does not recognize SID[i], or SID[i] is not valid or has already been used, the passive opener SHOULD inspect any other ENO suboptions in hopes of negotiating a fresh key exchange as described in Section 3.4.

When two hosts have previously negotiated a tcpcrypt session, either host may initiate session resumption regardless of which host was the active opener or played the "A" role in the previous session. However, a given host must either encrypt with k_ab for all sessions derived from the same pre-session seed, or k_ba. Thus, which keys a host uses to send segments depends only whether the host played the

"A" or "B" role in the initial session that used ss[0]; it is not affected by which host was the active opener transmitting the SYN segment containing a resumption suboption.

A host MUST ignore a resumption suboption if it has previously sent or received one with the same SID[i]. In the event that two hosts simultaneously send SYN segments to each other with the same SID[i], but the two segments are not part of a simultaneous open, both connections will have to revert to public key cryptography. To avoid this limitation, implementations MAY choose to implement session caching such that a given pre-session key is only good for either passive or active opens at the same host, not both.

In the case of simultaneous open where TCP-ENO is able to establish asymmetric roles, two hosts that simultaneously send SYN segments with resumption suboptions containing the same SID[i] may resume the associated session.

Implementations that perform session caching MUST provide a means for applications to control session caching, including flushing cached session secrets associated with an ESTABLISHED connection or disabling the use of caching for a particular connection.

3.6. Data encryption and authentication

Following key exchange, all further communication in a tcpcrypt-enabled connection is carried out within delimited _application frames_ that are encrypted and authenticated using the agreed keys.

This protection is provided via algorithms for Authenticated Encryption with Associated Data (AEAD). The particular algorithms that may be used are listed in Table 2. One algorithm is selected during the negotiation described in Section 3.4.

The format of an application frame is specified in Section 4.2. A sending host breaks its stream of application data into a series of chunks. Each chunk is placed in the "data" portion of a frame’s "plaintext" value, which is then encrypted to yield the frame’s "ciphertext" field. Chunks must be small enough that the ciphertext (slightly longer than the plaintext) has length less than 2^16 bytes.

An "associated data" value (see Section 4.2.2) is constructed for the frame. It contains the frame’s "control" field and the length of the ciphertext.

A "frame nonce" value (see Section 4.2.3) is also constructed for the frame (but not explicitly transmitted), containing an "offset" field whose integer value is the byte-offset of the beginning of the

current application frame in the underlying TCP datastream. (That is, the offset in the framing stream, not the plaintext application stream.) As the security of the AEAD algorithm depends on this nonce being used to encrypt at most one distinct plaintext value, an implementation MUST NOT ever transmit distinct frames at the same location in the underlying TCP datastream.

With reference to the "AEAD Interface" described in Section 2 of [RFC5116], tcpcrypt invokes the AEAD algorithm with the secret key "K" set to k_ab or k_ba, according to the host's role as described in Section 3.4. The plaintext value serves as "P", the associated data as "A", and the frame nonce as "N". The output of the encryption operation, "C", is transmitted in the frame's "ciphertext" field.

When a frame is received, tcpcrypt reconstructs the associated data and frame nonce values (the former contains only data sent in the clear, and the latter is implicit in the TCP stream), and provides these and the ciphertext value to the the AEAD decryption operation. The output of this operation is either "P", a plaintext value, or the special symbol FAIL. In the latter case, the implementation MAY either ignore the frame or terminate the connection.

3.7. TCP header protection

The "ciphertext" field of the application frame contains protected versions of certain TCP header values.

When "URGp" is set, the "urgent" value indicates an offset from the current frame's beginning offset; the sum of these offsets gives the index of the last byte of urgent data in the application datastream.

When "FINp" is set, it indicates that the sender will send no more application data after this frame. A receiver MUST ignore the TCP FIN flag and instead wait for "FINp" to signal to the local application that the stream is complete.

3.8. Re-keying

Re-keying allows hosts to wipe from memory keys that could decrypt previously transmitted segments. It also allows the use of AEAD ciphers that can securely encrypt only a bounded number of messages under a given key.

We refer to the two encryption keys (k_ab, k_ba) as a _key-set_. We refer to the key-set generated by mk[i] as the key-set with _generation number_ "i" within a session. Each host maintains a _current generation number_ that it uses to encrypt outgoing frames. Initially, the two hosts have current generation number 0.

When a host has just incremented its current generation number and has used the new key-set for the first time to encrypt an outgoing frame, it MUST set the frame's "rekey" field (see Section 4.2) to 1. It MUST set this field to zero in all other cases.

A host MAY increment its generation number beyond the highest generation it knows the other side to be using. We call this action _initiating re-keying_.

A host SHOULD NOT initiate more than one concurrent re-key operation if it has no data to send.

On receipt, a host increments its record of the remote host's current generation number if and only if the "rekey" field is set to 1.

If a received frame's generation number is greater than the receiver's current generation number, the receiver MUST immediately increment its current generation number to match. After incrementing its generation number, if the receiver does not have any application data to send, it MUST send an empty application frame with the "rekey" field set to 1.

When retransmitting, implementations must always transmit the same bytes for the same TCP sequence numbers. Thus, a frame in a retransmitted segment MUST always be encrypted with the same key as when it was originally transmitted.

Implementations SHOULD delete older-generation keys from memory once they have received all frames they will need to decrypt with the old keys and have encrypted all outgoing frames under the old keys.

3.9. Keep-alive

Many hosts implement TCP Keep-Alives [RFC1122] as an option for applications to ensure that the other end of a TCP connection still exists even when there is no data to be sent. A TCP Keep-Alive segment carries a sequence number one prior to the beginning of the send window, and may carry one byte of "garbage" data. Such a segment causes the remote side to send an acknowledgment.

Unfortunately, tcpcrypt cannot cryptographically verify Keep-Alive acknowledgments. Hence, an attacker could prolong the existence of a session at one host after the other end of the connection no longer exists. (Such an attack might prevent a process with sensitive data from exiting, giving an attacker more time to compromise a host and extract the sensitive data.)

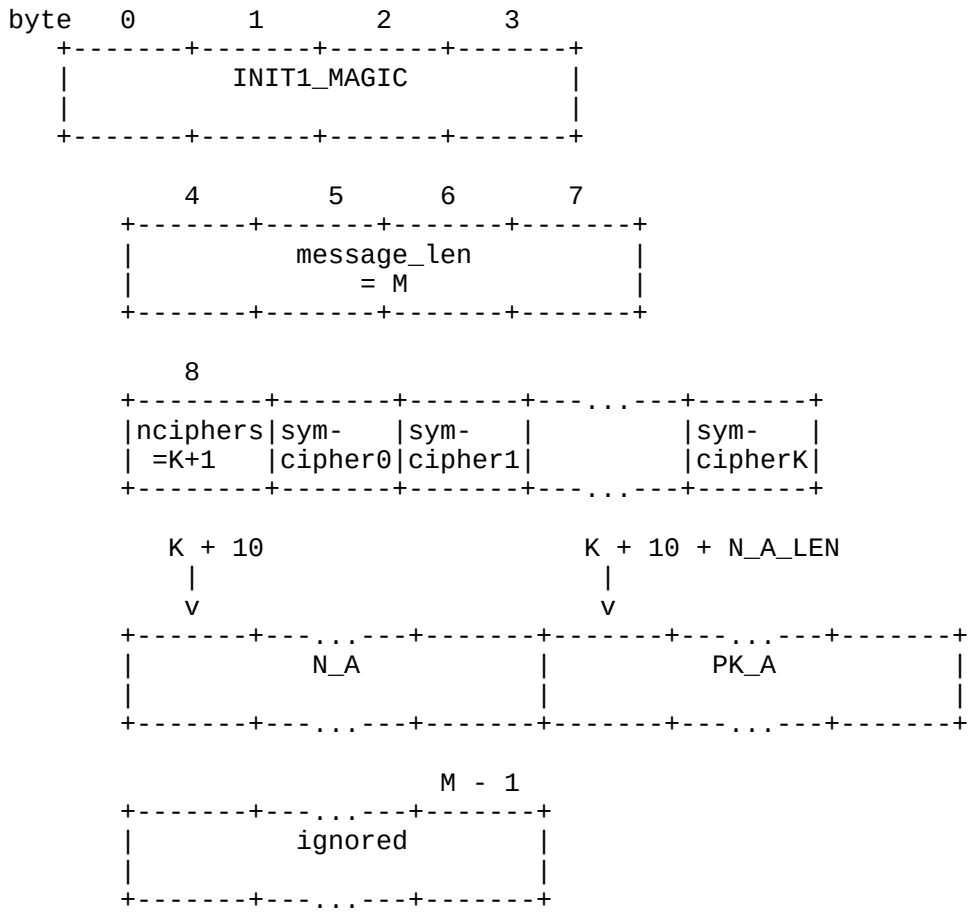
Instead of TCP Keep-Alives, tcpcrypt implementations SHOULD employ the re-keying mechanism to stimulate the remote host to send verifiably fresh and authentic data. When required, a host SHOULD probe the liveness of its peer by initiating re-keying as described in Section 3.8, and then transmitting a new frame (with zero-length application data if necessary). A host receiving a frame whose key generation number is greater than its current generation number MUST increment its current generation number and MUST immediately transmit a new frame (with zero-length application data, if necessary).

4. Encodings

This section provides byte-level encodings for values transmitted or computed by the protocol.

4.1. Key exchange messages

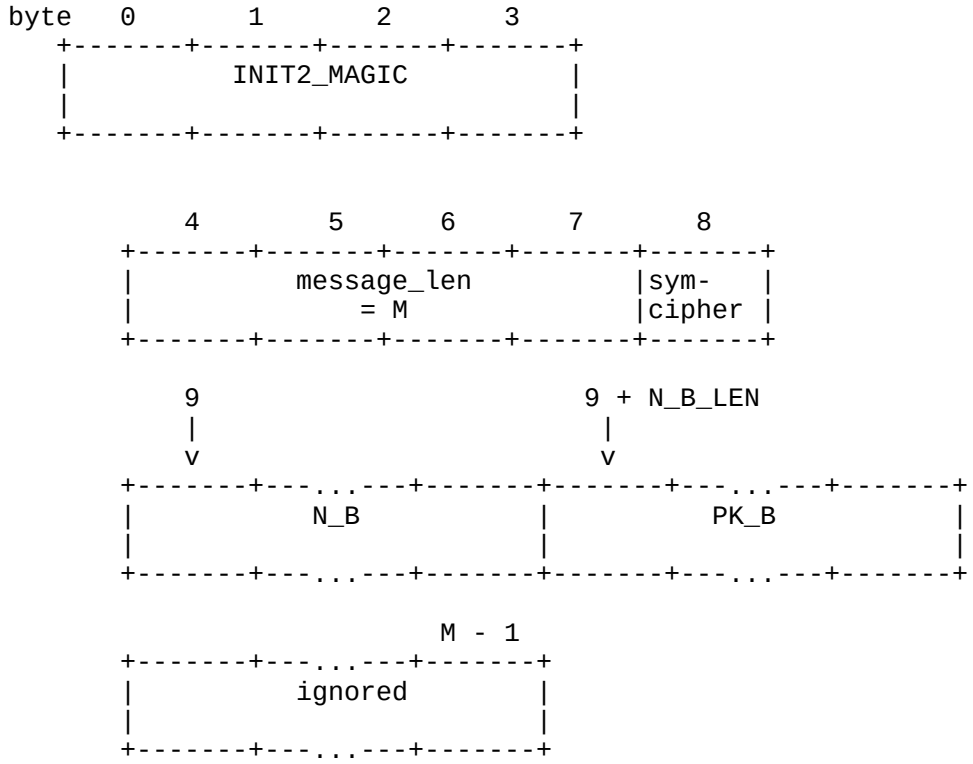
The INIT1 message has the following encoding:



The constant INIT1_MAGIC is defined in Table 3. The four-byte field "message_len" gives the length of the entire INIT1 message, encoded as a big-endian integer. The "nciphers" field contains an integer value that specifies the number of one-byte symmetric-cipher identifiers that follow. The "sym-cipher" bytes identify cryptographic algorithms in Table 2. The length N_A_LEN and the length of PK_A are both determined by the negotiated key-agreement scheme, as shown in Figure 3.

When sending INIT1, implementations of this protocol MUST omit the field "ignored"; that is, they must construct the message such that its end, as determined by "message_len", coincides with the end of the PK_A field. When receiving INIT1, however, implementations MUST permit and ignore any bytes following PK_A.

The INIT2 message has the following encoding:

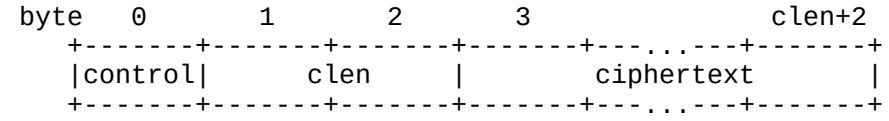


The constant INIT2_MAGIC is defined in Table 3. The four-byte field "message_len" gives the length of the entire INIT2 message, encoded as a big-endian integer. The "sym-cipher" value is a selection from the symmetric-cipher identifiers in the previously-received INIT1 message. The length N_B_LEN and the length of PK_B are both determined by the negotiated key-agreement scheme, as shown in Figure 3.

When sending INIT2, implementations of this protocol MUST omit the field "ignored"; that is, they must construct the message such that its end, as determined by "message_len", coincides with the end of the PK_B field. When receiving INIT2, however, implementations MUST permit and ignore any bytes following PK_B.

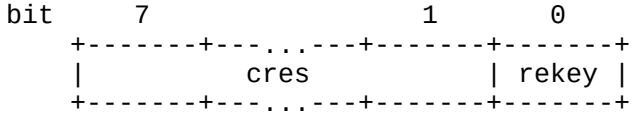
4.2. Application frames

An _application frame_ comprises a control byte and a length-prefixed ciphertext value:



The field "clen" is an integer in big-endian format and gives the length of the "ciphertext" field.

The byte "control" has this structure:

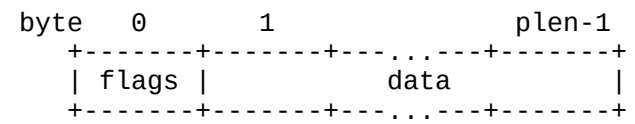


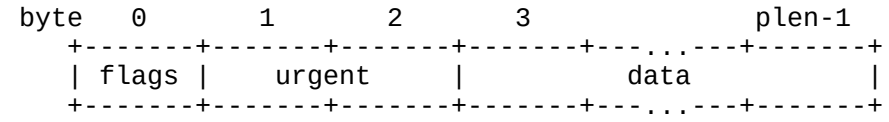
The seven-bit field "cres" is reserved; implementations MUST set these bits to zero when sending, and MUST ignore them when receiving.

The use of the "rekey" field is described in Section 3.8.

4.2.1. Plaintext

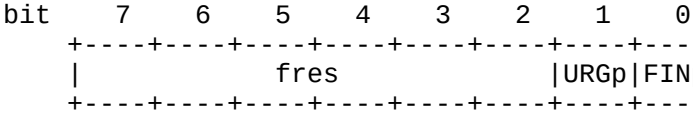
The "ciphertext" field is the result of applying the negotiated authenticated-encryption algorithm to a "plaintext" value, which has one of these two formats:





(Note that "clen" will generally be greater than "plen", as the authenticated-encryption scheme attaches an integrity "tag" to the encrypted input.)

The "flags" byte has this structure:



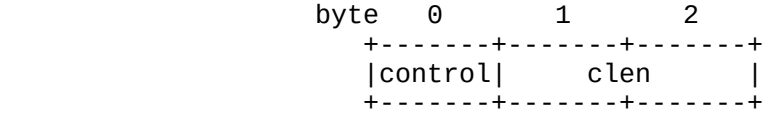
The six-bit value "fres" is reserved; implementations MUST set these six bits to zero when sending, and MUST ignore them when receiving.

When the "URGp" bit is set, it indicates that the "urgent" field is present, and thus that the plaintext value has the second structure variant above; otherwise the first variant is used.

The meaning of "urgent" and of the flag bits is described in Section 3.7.

4.2.2. Associated data

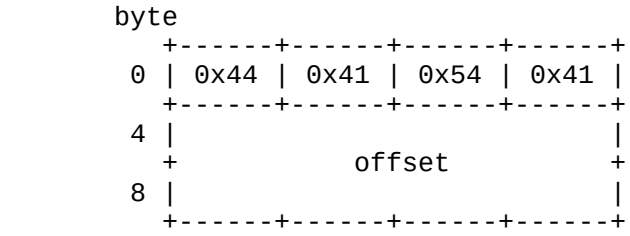
An application frame's "associated data" (which is supplied to the AEAD algorithm when decrypting the ciphertext and verifying the frame's integrity) has this format:



It contains the same values as the frame's "control" and "clen" fields.

4.2.3. Frame nonce

Lastly, a "frame nonce" (provided as input to the AEAD algorithm) has this format:



The 8-byte "offset" field contains an integer in big-endian format. Its value is specified in Section 3.6.

5. API extensions

Applications aware of tcpcrypt will need an API for interacting with the protocol. They can do so if implementations provide the recommended API for TCP-ENO. This section recommends several additions to that API, described in the style of socket options. However, these recommendations are non-normative:

The following options is read-only:

TCP_CRYPT_CONF: Returns the one-byte authenticated encryption algorithm in use by the connection (as specified in Table 2).

The following option is write-only:

TCP_CRYPT_CACHE_FLUSH: Setting this option to non-zero wipes cached session keys as specified in Section 3.5. Useful if application-level authentication discovers a man in the middle attack, to prevent the next connection from using session caching.

The following options should be readable and writable:

TCP_CRYPT_ACONF: Set of allowed symmetric ciphers and message authentication codes this host advertises in INIT1 messages.

TCP_CRYPT_BCONF: Order of preference of symmetric ciphers.

Finally, system administrators must be able to set the following system-wide parameters:

- Default TCP_CRYPT_ACONF value
- Default TCP_CRYPT_BCONF value
- Types, key lengths, and regeneration intervals of local host’s short-lived public keys for implementations that do not use fresh ECDH parameters for each connection.

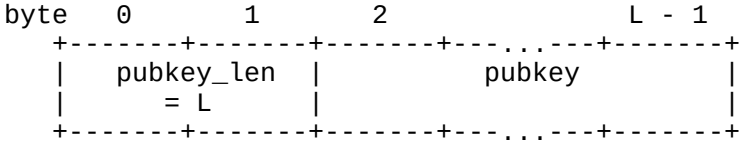
6. Key agreement schemes

The encryption spec negotiated via TCP-ENO may indicate the use of one of these key-agreement schemes:

Encryption spec (cs)	Key-agreement scheme
TCPCRYPT_ECDHE_P256	Cipher: ECDHE-P256 Extract: HKDF-Extract-SHA256 CPRF: HKDF-Expand-SHA256 N_A_LEN: 32 bytes N_B_LEN: 32 bytes K_LEN: 32 bytes
TCPCRYPT_ECDHE_P521	Cipher: ECDHE-P521 Extract: HKDF-Extract-SHA256 CPRF: HKDF-Expand-SHA256 N_A_LEN: 32 bytes N_B_LEN: 32 bytes K_LEN: 32 bytes
TCPCRYPT_ECDHE_Curve25519	Cipher: ECDHE-Curve25519 Extract: HKDF-Extract-SHA256 CPRF: HKDF-Expand-SHA256 N_A_LEN: 32 bytes N_B_LEN: 32 bytes K_LEN: 32 bytes

Figure 3: Key agreement schemes

Ciphers ECDHE-P256 and ECDHE-P521 employ the ECSVDP-DH secret value derivation primitive defined in [ieee1363]. The named curves are defined in [nist-dss]. When the public-key values PK_A and PK_B are transmitted as described in Section 4.1, they are encoded with the "Elliptic Curve Point to Octet String Conversion Primitive" described in Section E.2.3 of [ieee1363], and are prefixed by a two-byte length in big-endian format:



Implementations SHOULD encode these "pubkey" values in "compressed format", and MUST accept values encoded in "compressed", "uncompressed" or "hybrid" formats.

The ECDHE-Curve25519 cipher uses the X25519 function described in [I-D.irtf-cfrg-curves]. When using this cipher, public-key values

PK_A and PK_B are transmitted directly as 32-byte values (with no length prefix).

A tcpcrypt implementation MUST support at least the schemes TCPCRYPT_ECDHE_P256 and TCPCRYPT_ECDHE_P521, although system administrators need not enable them.

7. AEAD algorithms

Specifiers and key-lengths for AEAD algorithms are given in Table 2. The algorithms AEAD_AES_128_GCM and AEAD_AES_256_GCM are specified in [RFC5116]. The algorithm AEAD_CHACHA20_POLY1305 is specified in [RFC7539].

8. Acknowledgments

This work was funded by gifts from Intel (to Brad Karp) and from Google, by NSF award CNS-0716806 (A Clean-Slate Infrastructure for Information Flow Control), and by DARPA CRASH under contract #N66001-10-2-4088.

9. IANA Considerations

Tcpcrypt's spec identifiers ("cs" values) will need to be added to IANA's ENO suboption registry, as follows:

cs	Spec name	Meaning
0x20	TCPCRYPT_RESUME	tcpcrypt session resumption
0x21	TCPCRYPT_ECDHE_P256	tcpcrypt with ECDHE-P256
0x22	TCPCRYPT_ECDHE_P521	tcpcrypt with ECDHE-P521
0x23	TCPCRYPT_ECDHE_Curve25519	tcpcrypt with ECDHE-Curve25519

Table 1: cs values for use with tcpcrypt

A "tcpcrypt AEAD parameter" registry needs to be maintained by IANA as per the following table. The use of encryption is described in Section 3.6.

AEAD Algorithm	Key Length	sym-cipher
AEAD_AES_128_GCM	16 bytes	0x01
AEAD_AES_256_GCM	32 bytes	0x02
AEAD_CHACHA20_POLY1305	32 bytes	0x10

Table 2: Authenticated-encryption algorithms corresponding to sym-cipher specifiers in INIT1 and INIT2 messages.

10. Security considerations

It is worth reiterating just how crucial both the quality and quantity of randomness are to tcpcrypt's security. Most implementations will rely on system-wide pseudo-random generators seeded from hardware events and a seed carried over from the previous boot. Once a pseudo-random generator has been properly seeded, it can generate effectively arbitrary amounts of pseudo-random data. However, until a pseudo-random generator has been seeded with sufficient entropy, not only will tcpcrypt be insecure, it will reveal information that further weakens the security of the pseudo-random generator, potentially harming other applications. In the absence of secure hardware random generators, implementations MUST disable tcpcrypt after rebooting until the pseudo-random generator has been reseeded (usually by a bootup script) or sufficient entropy has been gathered.

Tcpcrypt guarantees that no man-in-the-middle attacks occurred if Session IDs match on both ends of a connection, unless the attacker has broken the underlying cryptographic primitives (e.g., ECDH). A proof has been published [tcpcrypt].

All of the security considerations of TCP-ENO apply to tcpcrypt. In particular, tcpcrypt does not protect against active eavesdroppers unless applications authenticate the Session ID.

To gain middlebox compatibility, tcpcrypt does not protect TCP headers. Hence, the protocol is vulnerable to denial-of-service from off-path attackers. Possible attacks include desynchronizing the underlying TCP stream, injecting RST packets, and forging or suppressing rekey bits. These attacks will cause a tcpcrypt connection to hang or fail with an error. Implementations MUST give higher-level software a way to distinguish such errors from a clean end-of-stream (indicated by an authenticated "FINp" bit) so that applications can avoid semantic truncation attacks.

Similarly, tcpcrypt does not have a key confirmation step. Hence, an active attacker can cause a connection to hang, though this is possible even without tcpcrypt by altering sequence and ack numbers.

Tcpcrypt uses short-lived public key parameters to provide forward secrecy. All currently specified key agreement schemes involve ECDHE-based key agreement, meaning a new key can be chosen for each connection. If implementations reuse these parameters, they SHOULD limit the lifetime of the private parameters, ideally to no more than two minutes.

Attackers cannot force passive openers to move forward in their session caching chain without guessing the content of the resumption suboption, which will be hard without key knowledge.

11. References

11.1. Normative References

[I-D.ietf-tcpinc-tcpeno]
Bittau, A., Boneh, D., Giffin, D., Handley, M., Mazieres, D., and E. Smith, "TCP-ENO: Encryption Negotiation Option", draft-ietf-tcpinc-tcpeno-00 (work in progress), September 2015.

[I-D.irtf-cfrg-curves]
Langley, A. and M. Hamburg, "Elliptic Curves for Security", draft-irtf-cfrg-curves-10 (work in progress), October 2015.

[ieee1363]
"IEEE Standard Specifications for Public-Key Cryptography (IEEE Std 1363-2000)", 2000.

[nist-dss]
"Digital Signature Standard, FIPS 186-2", 2000.

[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>.

[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>.

[RFC7539] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 7539, DOI 10.17487/RFC7539, May 2015, <http://www.rfc-editor.org/info/rfc7539>.

11.2. Informative References

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <http://www.rfc-editor.org/info/rfc1122>.

[tcpcrypt] Bittau, A., Hamburg, M., Handley, M., Mazieres, D., and D. Boneh, "The case for ubiquitous transport-level encryption", USENIX Security , 2010.

Appendix A. Protocol constant values

Value	Name
0x01	CONST_NEXTK
0x02	CONST_SESSID
0x03	CONST_REKEY
0x04	CONST_KEY_A
0x05	CONST_KEY_B
0x15101a0e	INIT1_MAGIC
0x097105e0	INIT2_MAGIC

Table 3: Protocol constants

Authors' Addresses

Andrea Bittau
Stanford University
353 Serra Mall, Room 288
Stanford, CA 94305
US

Email: bittau@cs.stanford.edu

Dan Boneh
Stanford University
353 Serra Mall, Room 475
Stanford, CA 94305
US

Email: dabo@cs.stanford.edu

Daniel B. Giffin
Stanford University
353 Serra Mall, Room 288
Stanford, CA 94305
US

Email: dbg@scs.stanford.edu

Mike Hamburg
Stanford University
353 Serra Mall, Room 475
Stanford, CA 94305
US

Email: mike@shiftleft.org

Mark Handley
University College London
Gower St.
London WC1E 6BT
UK

Email: M.Handley@cs.ucl.ac.uk

David Mazieres
Stanford University
353 Serra Mall, Room 290
Stanford, CA 94305
US

Email: dm@uun.org

Quinn Slack
Stanford University
353 Serra Mall, Room 288
Stanford, CA 94305
US

Email: sqs@cs.stanford.edu

Eric W. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304
US

Email: eric.smith@kestrel.edu

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: June 14, 2019

A. Bittau
Google
D. Giffin
Stanford University
M. Handley
University College London
D. Mazieres
Stanford University
Q. Slack
Sourcegraph
E. Smith
Kestrel Institute

December 11, 2018

Cryptographic protection of TCP Streams (tcpcrypt)
draft-ietf-tcpinc-tcpcrypt-15

Abstract

This document specifies tcpcrypt, a TCP encryption protocol designed for use in conjunction with the TCP Encryption Negotiation Option (TCP-ENO). Tcpcrypt coexists with middleboxes by tolerating resegmentation, NATs, and other manipulations of the TCP header. The protocol is self-contained and specifically tailored to TCP implementations, which often reside in kernels or other environments in which large external software dependencies can be undesirable. Because the size of TCP options is limited, the protocol requires one additional one-way message latency to perform key exchange before application data can be transmitted. However, the extra latency can be avoided between two hosts that have recently established a previous tcpcrypt connection.

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 June 14, 2019.

Table of Contents

1. Requirements Language

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.

2. Introduction

This document describes tcpcrypt, an extension to TCP for cryptographic protection of session data. Tcpcrypt was designed to meet the following goals:

- Meet the requirements of the TCP Encryption Negotiation Option (TCP-ENO) [I-D.ietf-tcpinc-tcpeno] for protecting connection data.
- Be amenable to small, self-contained implementations inside TCP stacks.
- Minimize additional latency at connection startup.
- As much as possible, prevent connection failure in the presence of NATs and other middleboxes that might normalize traffic or otherwise manipulate TCP segments.
- Operate independently of IP addresses, making it possible to authenticate resumed sessions efficiently even when either end changes IP address.

A companion document [I-D.ietf-tcpinc-api] describes recommended interfaces for configuring certain parameters of this protocol.

3. Encryption Protocol

This section describes the operation of the tcpcrypt protocol. The wire format of all messages is specified in Section 4.

3.1. Cryptographic Algorithms

Setting up a tcpcrypt connection employs three types of cryptographic algorithms:

- A _key agreement scheme_ is used with a short-lived public key to agree upon a shared secret.

- An _extract function_ is used to generate a pseudo-random key (PRK) from some initial keying material produced by the key agreement scheme. The notation Extract(S, IKM) denotes the output of the extract function with salt S and initial keying material IKM.

- A _collision-resistant pseudo-random function (CPRF)_ is used to generate multiple cryptographic keys from a pseudo-random key, typically the output of the extract function. The CPRF produces an arbitrary amount of Output Keying Material (OKM), and we use the notation CPRF(K, CONST, L) to designate the first L bytes of the OKM produced by the CPRF when parameterized by key K and the constant CONST.

The Extract and CPRF functions used by the tcpcrypt variants defined in this document are the Extract and Expand functions of HKDF [RFC5869], which is built on HMAC [RFC2104]. These are defined as follows in terms of the function "HMAC-Hash(key, value)" for a negotiated "Hash" function such as SHA-256; the symbol "|" denotes concatenation, and the counter concatenated to the right of CONST occupies a single octet.

```
HKDF-Extract(salt, IKM) -> PRK
   PRK = HMAC-Hash(salt, IKM)

HKDF-Expand(PRK, CONST, L) -> OKM
   T(0) = empty string (zero length)
   T(1) = HMAC-Hash(PRK, T(0) | CONST | 0x01)
   T(2) = HMAC-Hash(PRK, T(1) | CONST | 0x02)
   T(3) = HMAC-Hash(PRK, T(2) | CONST | 0x03)
   ...

   OKM  = first L octets of T(1) | T(2) | T(3) | ...
   where L <= 255*OutputLength(Hash)
```

Figure 1: HKDF functions used for key derivation

Lastly, once tcpcrypt has been successfully set up and encryption keys have been derived, an algorithm for Authenticated Encryption with Associated Data (AEAD) is used to protect the confidentiality and integrity of all transmitted application data. AEAD algorithms use a single key to encrypt their input data and also to generate a cryptographic tag to accompany the resulting ciphertext; when decryption is performed, the tag allows authentication of the encrypted data and of optional, associated plaintext data.

3.2. Protocol Negotiation

Tcpcrypt depends on TCP-ENO [I-D.ietf-tcpinc-tcpeno] to negotiate whether encryption will be enabled for a connection, and also which key-agreement scheme to use. TCP-ENO negotiates the use of a particular TCP encryption protocol or _TEP_ by including protocol identifiers in ENO suboptions. This document associates four TEP identifiers with the tcpcrypt protocol, as listed in Table 4 in Section 7. Each identifier indicates the use of a particular key-agreement scheme, with an associated CPRF and length parameter. Future standards can associate additional TEP identifiers with tcpcrypt, following the assignment policy specified by TCP-ENO.

An active opener that wishes to negotiate the use of tcpcrypt includes an ENO option in its SYN segment. That option includes suboptions with tcpcrypt TEP identifiers indicating the key-agreement schemes it is willing to enable. The active opener MAY additionally include suboptions indicating support for encryption protocols other than tcpcrypt, as well as global suboptions as specified by TCP-ENO.

If a passive opener receives an ENO option including tcpcrypt TEPs it supports, it MAY then attach an ENO option to its SYN-ACK segment, including solely the TEP it wishes to enable.

To establish distinct roles for the two hosts in each connection, tcpcrypt depends on the role-negotiation mechanism of TCP-ENO. As one result of the negotiation process, TCP-ENO assigns hosts unique roles abstractly called "A" at one end of the connection and "B" at the other. Generally, an active opener plays the "A" role and a passive opener plays the "B" role, but in the case of simultaneous open, an additional mechanism breaks the symmetry and assigns a distinct role to each host. TCP-ENO uses the terms "host A" and "host B" to identify each end of a connection uniquely, and this document employs those terms in the same way.

An ENO suboption includes a flag "v" which indicates the presence of associated, variable-length data. In order to propose fresh key agreement with a particular tcpcrypt TEP, a host sends a one-byte suboption containing the TEP identifier and "v = 0". In order to propose session resumption (described further below) with a particular TEP, a host sends a variable-length suboption containing the TEP identifier, the flag "v = 1", an identifier derived from a session secret previously negotiated with the same host and the same TEP, and a nonce.

Once two hosts have exchanged SYN segments, TCP-ENO defines the _negotiated TEP_ to be the last valid TEP identifier in the SYN segment of host B (that is, the passive opener in the absence of

simultaneous open) that also occurs in that of host A. If there is no such TEP, hosts MUST disable TCP-ENO and tcpcrypt.

If the negotiated TEP was sent by host B with "v = 0", it means that fresh key agreement will be performed as described below in Section 3.3. If, on the other hand, host B sent the TEP with "v = 1" and both hosts sent appropriate resumption identifiers in their suboption data, then the key-exchange messages will be omitted in favor of determining keys via session resumption as described in Section 3.5. With session resumption, protected application data MAY be sent immediately as detailed in Section 3.6.

Note that the negotiated TEP is determined without reference to the "v" bits in ENO suboptions, so if host A offers resumption with a particular TEP and host B replies with a non-resumption suboption with the same TEP, that could become the negotiated TEP and fresh key agreement will be performed. That is, sending a resumption suboption also implies willingness to perform fresh key agreement with the indicated TEP.

As REQUIRED by TCP-ENO, once a host has both sent and received an ACK segment containing a valid ENO option, encryption MUST be enabled and plaintext application data MUST NOT ever be exchanged on the connection. If the negotiated TEP is among those listed in Table 4, a host MUST follow the protocol described in this document.

3.3. Key Exchange

Following successful negotiation of a tcpcrypt TEP, all further signaling is performed in the Data portion of TCP segments. Except when resumption was negotiated (described below in Section 3.5), the two hosts perform key exchange through two messages, "Init1" and "Init2", at the start of the data streams of host A and host B, respectively. These messages MAY span multiple TCP segments and need not end at a segment boundary. However, the segment containing the last byte of an "Init1" or "Init2" message MUST have TCP's push flag (PSH) set.

The key exchange protocol, in abstract, proceeds as follows:

```
A -> B:  Init1 = { INIT1_MAGIC, sym_cipher_list, N_A, Pub_A }
B -> A:  Init2 = { INIT2_MAGIC, sym_cipher, N_B, Pub_B }
```

The concrete format of these messages is specified in Section 4.1.

The parameters are defined as follows:

- "INIT1_MAGIC", "INIT2_MAGIC": constants defined in Section 4.3.

- "sym_cipher_list": a list of identifiers of symmetric ciphers (AEAD algorithms) acceptable to host A. These are specified in Table 5 in Section 7.

- "sym_cipher": the symmetric cipher selected by host B from the "sym_cipher_list" sent by host A.

- "N_A", "N_B": nonces chosen at random by hosts A and B, respectively.

- "Pub_A", "Pub_B": ephemeral public keys for hosts A and B, respectively. These, as well as their corresponding private keys, are short-lived values that MUST be refreshed frequently. The private keys SHOULD NOT ever be written to persistent storage. The security risks associated with the storage of these keys are discussed in Section 8.

If a host receives an ephemeral public key from its peer and a key-validation step fails (see Section 5), it MUST abort the connection and raise an error condition distinct from the end-of-file condition.

The ephemeral secret "ES" is the result of the key-agreement algorithm (see Section 5) indicated by the negotiated TEP. The inputs to the algorithm are the local host's ephemeral private key and the remote host's ephemeral public key. For example, host A would compute "ES" using its own private key (not transmitted) and host B's public key, "Pub_B".

The two sides then compute a pseudo-random key "PRK", from which all session secrets are derived, as follows:

```
PRK = Extract(N_A, eno-transcript | Init1 | Init2 | ES)
```

Above, "|" denotes concatenation; "eno-transcript" is the protocol-negotiation transcript defined in Section 4.8 of [I-D.ietf-tcpinc-tcpeno]; and "Init1" and "Init2" are the transmitted encodings of the messages described in Section 4.1.

A series of "session secrets" are computed from "PRK" as follows:

```
ss[0] = PRK
ss[i] = CPRF(ss[i-1], CONST_NEXTK, K_LEN)
```

The value "ss[0]" is used to generate all key material for the current connection. The values "ss[i]" for "i > 0" are used by session resumption to avoid public key cryptography when establishing subsequent connections between the same two hosts, as described later in Section 3.5. The "CONST_*" values are constants defined in

Section 4.3. The length "K_LEN" depends on the tcpcrypt TEP in use, and is specified in Section 5.

Given a session secret "ss[i]", the two sides compute a series of master keys as follows:

```
mk[0] = CPRF(ss[i], CONST_REKEY | sn[i], K_LEN)
mk[j] = CPRF(mk[j-1], CONST_REKEY, K_LEN)
```

The process of advancing through the series of master keys is described in Section 3.8. The values "sn[i]" are "session nonces." For the initial session with "i = 0", the session nonce is zero bytes long. The values for subsequent sessions are derived from fresh connection data as described in Section 3.5.

Finally, each master key "mk[j]" is used to generate traffic keys for protecting application data using authenticated encryption:

```
k_ab[j] = CPRF(mk[j], CONST_KEY_A, ae_key_len + ae_nonce_len)
k_ba[j] = CPRF(mk[j], CONST_KEY_B, ae_key_len + ae_nonce_len)
```

In the first session derived from fresh key-agreement, traffic keys "k_ab[j]" are used by host A to encrypt and host B to decrypt, while keys "k_ba[j]" are used by host B to encrypt and host A to decrypt. In a resumed session, as described more thoroughly below in Section 3.5, each host uses the keys in the same way as it did in the original session, regardless of its role in the current session: for example, if a host played role "A" in the first session, it will use keys "k_ab[j]" to encrypt in each derived session.

The values "ae_key_len" and "ae_nonce_len" depend on the authenticated-encryption algorithm selected, and are given in Table 3 in Section 6. The algorithm uses the first "ae_key_len" bytes of each traffic key as an authenticated-encryption key, and the following "ae_nonce_len" bytes as a "nonce randomizer".

Implementations SHOULD provide an interface allowing the user to specify, for a particular connection, the set of AEAD algorithms to advertize in "sym_cipher_list" (when playing role "A") and also the order of preference to use when selecting an algorithm from those offered (when playing role "B"). A companion document [I-D.ietf-tcpinc-api] describes recommended interfaces for this purpose.

After host B sends "Init2" or host A receives it, that host MAY immediately begin transmitting protected application data as described in Section 3.6.

If host A receives "Init2" with a "sym_cipher" value that was not present in the "sym_cipher_list" it previously transmitted in "Init1", it MUST abort the connection and raise an error condition distinct from the end-of-file condition.

Throughout this document, to "abort the connection" means to issue the "Abort" command as described in [RFC0793], Section 3.8. That is, the TCP connection is destroyed, RESET is transmitted, and the local user is alerted to the abort event.

3.4. Session ID

TCP-ENO requires each TEP to define a _session ID_ value that uniquely identifies each encrypted connection.

A tcpcrypt session ID begins with the byte transmitted by host B that contains the negotiated TEP identifier along with the "v" bit. The remainder of the ID is derived from the session secret and session nonce, as follows:

```
session_id[i] = TEP-byte | CPRF(ss[i], CONST_SESSID | sn[i], K_LEN)
```

Again, the length "K_LEN" depends on the TEP, and is specified in Section 5.

3.5. Session Resumption

If two hosts have previously negotiated a session with secret "ss[i-1]", they can establish a new connection without public-key operations using "ss[i]", the next session secret in the sequence derived from the original PRK.

A host signals willingness to resume with a particular session secret by sending a SYN segment with a resumption suboption: that is, an ENO suboption containing the negotiated TEP identifier of the previous session, half of the "resumption identifier" for the new session, and a "resumption nonce".

The resumption nonce MUST have a minimum length of zero bytes and maximum length of eight bytes. The value MUST be chosen randomly or using a mechanism that guarantees uniqueness even in the face of virtual machine cloning or other re-execution of the same session. An attacker who can force either side of a connection to reuse a session secret with the same nonce will completely break the security of tcpcrypt. Reuse of session secrets is possible in the event of virtual machine cloning or reuse of system-level hibernation state. Implementations SHOULD provide an API through which to set the

resumption nonce length, and MUST default to eight bytes if they cannot prohibit the reuse of session secrets.

The resumption identifier is calculated from a session secret "ss[i]" as follows:

resume[i] = CPRF(ss[i], CONST_RESUME, 18)

To name a session for resumption, a host sends either the first or second half of the resumption identifier, according to the role it played in the original session with secret "ss[0]".

A host that originally played role "A" and wishes to resume from a cached session sends a suboption with the first half of the resumption identifier:

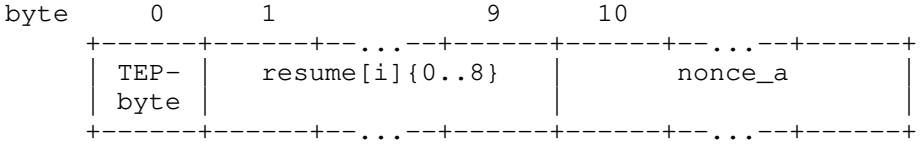


Figure 2: Resumption suboption sent when original role was "A". The TEP-byte contains a tcpcrypt TEP identifier and v = 1. The nonce value MUST have length between 0 and 8 bytes.

Similarly, a host that originally played role "B" sends a suboption with the second half of the resumption identifier:

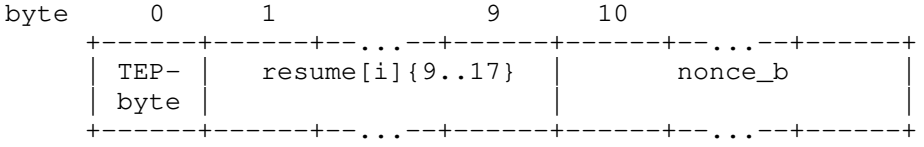


Figure 3: Resumption suboption sent when original role was "B". The TEP-byte contains a tcpcrypt TEP identifier and v = 1. The nonce value MUST have length between 0 and 8 bytes.

If a passive opener receives a resumption suboption containing an identifier-half that names a session secret that it has cached and the subobtion's TEP matches the TEP used in the previous session, it SHOULD (with exceptions specified below) agree to resume from the cached session by sending its own resumption suboption, which will contain the other half of the identifier. Otherwise, it MUST NOT agree to resumption.

If a passive opener does not agree to resumption with a particular TEP, it MAY either request fresh key exchange by responding with a non-resumption suboption using the same TEP, or else respond to any other received TEP suboption.

If a passive opener receives an ENO suboption with a TEP identifier and "v = 1", but the suboption data is less than 9 bytes in length, it MUST behave as if the same TEP had been sent with "v = 0". That is, the suboption MUST be interpreted as an offer to negotiate fresh key exchange with that TEP.

If an active opener sends a resumption suboption with a particular TEP and the appropriate half of a resumption identifier and then, in the same TCP handshake, receives a resumption suboption with the same TEP and an identifier-half that does not match that resumption identifier, it MUST ignore that suboption. In the typical case that this was the only ENO suboption received, this means the host MUST disable TCP-ENO and tcpcrypt: that is, it MUST NOT send any more ENO options and MUST NOT encrypt the connection.

When a host concludes that TCP-ENO negotiation has succeeded for some TEP that was received in a resumption suboption, it MUST then enable encryption with that TEP using the cached session secret. To do this, it first constructs "sn[i]" as follows:

$$sn[i] = nonce\_a \mid nonce\_b$$

Master keys are then computed from "s[i]" and "sn[i]" as described in Section 3.3, and application data encrypted as described in Section 3.6.

The session ID (Section 3.4) is constructed in the same way for resumed sessions as it is for fresh ones. In this case the first byte will always have "v = 1". The remainder of the ID is derived from the cached session secret and the session nonce that was generated during resumption.

In the case of simultaneous open where TCP-ENO is able to establish asymmetric roles, two hosts that simultaneously send SYN segments with compatible resumption suboptions MAY resume the associated session.

In a particular SYN segment, a host SHOULD NOT send more than one resumption suboption (because this consumes TCP option space and is unlikely to be a useful practice), and MUST NOT send more than one resumption suboption with the same TEP identifier. But in addition to any resumption suboptions, an active opener MAY include non-

resumption suboptions describing other TEPs it supports (in addition to the TEP in the resumption suboption).

After using the session secret "ss[i]" to compute "mk[0]", implementations SHOULD compute and cache "ss[i+1]" for possible use by a later session, then erase "ss[i]" from memory. Hosts MAY retain "ss[i+1]" until it is used or the memory needs to be reclaimed. Hosts SHOULD NOT write any session secrets to non-volatile storage.

When proposing resumption, the active opener MUST use the lowest value of "i" that has not already been used (successfully or not) to negotiate resumption with the same host and for the same original session secret "ss[0]".

A given session secret "ss[i]" MUST NOT be used to secure more than one TCP connection. To prevent this, a host MUST NOT resume with a session secret if it has ever enabled encryption in the past with the same secret, in either role. In the event that two hosts simultaneously send SYN segments to each other that propose resumption with the same session secret but the two segments are not part of a simultaneous open, both connections would need to revert to fresh key-exchange. To avoid this limitation, implementations MAY choose to implement session resumption such that all session secrets derived from a given "ss[0]" are used for either passive or active opens at the same host, not both.

If two hosts have previously negotiated a tcpcrypt session, either host MAY later initiate session resumption regardless of which host was the active opener or played the "A" role in the previous session.

However, a given host MUST either encrypt with keys "k_ab[j]" for all sessions derived from the same original session secret "ss[0]", or with keys "k_ba[j]". Thus, which keys a host uses to send segments is not affected by the role it plays in the current connection: it depends only on whether the host played the "A" or "B" role in the initial session.

Implementations that cache session secrets MUST provide a means for applications to control that caching. In particular, when an application requests a new TCP connection, it MUST have a way to specify two policies for the duration of the connection: 1) that resumption requests will be ignored, and thus fresh key exchange will be necessary; and 2) that no session secrets will be cached. (These policies can be specified independently or as a unit.) And for an established connection, an application MUST have a means to cause any cache state that was used in or resulted from establishing the connection to be flushed. A companion document [I-D.ietf-tcpinc-api] describes recommended interfaces for this purpose.

3.6. Data Encryption and Authentication

Following key exchange (or its omission via session resumption), all further communication in a tcpcrypt-enabled connection is carried out within delimited _encryption frames_ that are encrypted and authenticated using the agreed upon keys.

This protection is provided via algorithms for Authenticated Encryption with Associated Data (AEAD). The permitted algorithms are listed in Table 5 in Section 7. Additional algorithms can be specified in the future according to the policy in that section. One algorithm is selected during the negotiation described in Section 3.3. The lengths "ae_key_len" and "ae_nonce_len" associated with each algorithm are found in Table 3 in Section 6, together with requirements for which algorithms MUST be implemented.

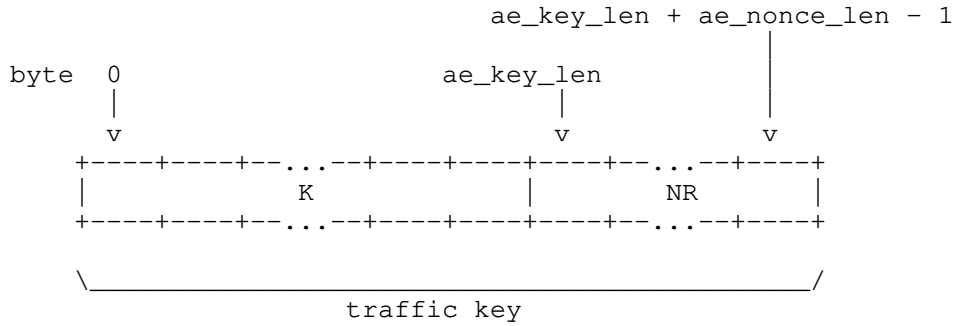
The format of an encryption frame is specified in Section 4.2. A sending host breaks its stream of application data into a series of chunks. Each chunk is placed in the "data" portion of a "plaintext" value, which is then encrypted to yield a frame's "ciphertext" field. Chunks MUST be small enough that the ciphertext (whose length depends on the AEAD cipher used, and is generally slightly longer than the plaintext) has length less than 2^16 bytes.

An "associated data" value (see Section 4.2.2) is constructed for the frame. It contains the frame's "control" field and the length of the ciphertext.

A "frame ID" value (see Section 4.2.3) is also constructed for the frame, but not explicitly transmitted. It contains a 64-bit "offset" field whose integer value is the zero-indexed byte offset of the beginning of the current encryption frame in the underlying TCP datastream. (That is, the offset in the framing stream, not the plaintext application stream.) The offset is then left-padded with zero-valued bytes to form a value of length "ae_nonce_len". Because it is strictly necessary for the security of the AEAD algorithms specified in this document, an implementation MUST NOT ever transmit distinct frames with the same frame ID value under the same encryption key. In particular, a retransmitted TCP segment MUST contain the same payload bytes for the same TCP sequence numbers, and a host MUST NOT transmit more than 2^64 bytes in the underlying TCP datastream (which would cause the "offset" field to wrap) before re-keying as decribed in Section 3.8.

With reference to the "AEAD Interface" described in Section 2 of [RFC5116], tcpcrypt invokes the AEAD algorithm with values taken from the traffic key "k_ab[j]" or "k_ba[j]" for some "j", according to the host's role as described in Section 3.3.

First, the traffic key is divided into two parts:



The first "ae_key_len" bytes of the traffic key provide the AEAD key "K", while the remaining "ae_nonce_len" bytes provide a "nonce randomizer" value "NR". The frame ID is then combined via bitwise exclusive-or with the nonce randomizer to yield "N", the AEAD nonce for the frame:

N = frame_ID XOR NR

The plaintext value serves as "P", and the associated data as "A". The output of the encryption operation, "C", is transmitted in the frame's "ciphertext" field.

When a frame is received, tcpcrypt reconstructs the associated data and frame ID values (the former contains only data sent in the clear, and the latter is implicit in the TCP stream), computes the nonce "N" as above, and provides these and the ciphertext value to the AEAD decryption operation. The output of this operation is either a plaintext value "P" or the special symbol FAIL. In the latter case, the implementation SHOULD abort the connection and raise an error condition distinct from the end-of-file condition. But if none of the TCP segment(s) containing the frame have been acknowledged and retransmission could potentially result in a valid frame, an implementation MAY instead drop these segments (and "renege" if they have been SACKed, according to [RFC2018] Section 8).

3.7. TCP Header Protection

The "ciphertext" field of the encryption frame contains protected versions of certain TCP header values.

When the "URGp" bit is set, the "urgent" value indicates an offset from the current frame's beginning offset; the sum of these offsets gives the index of the last byte of urgent data in the application datastream.

A sender MUST set the "FINp" bit on the last frame it sends in the connection (unless it aborts the connection), and MUST NOT set "FINp" on any other frame.

TCP sets the FIN flag when a sender has no more data, which with tcpcrypt means setting FIN on the segment containing the last byte of the last frame. However, a receiver MUST report the end-of-file condition to the connection's local user when and only when it receives a frame with the "FINp" bit set. If a host receives a segment with the TCP FIN flag set but the received datastream including this segment does not contain a frame with "FINp" set, the host SHOULD abort the connection and raise an error condition distinct from the end-of-file condition. But if there are unacknowledged segments whose retransmission could potentially result in a valid frame, the host MAY instead drop the segment with the TCP FIN flag set (and "renege" if it has been SACKed, according to [RFC2018] Section 8).

3.8. Re-Keying

Re-keying allows hosts to wipe from memory keys that could decrypt previously transmitted segments. It also allows the use of AEAD ciphers that can securely encrypt only a bounded number of messages under a given key.

As described above in Section 3.3, a master key "mk[j]" is used to generate two encryption keys "k_ab[j]" and "k_ba[j]". We refer to these as a _key-set_ with _generation number_ "j". Each host maintains a _local generation number_ that determines which key-set it uses to encrypt outgoing frames, and a _remote generation number_ equal to the highest generation used in frames received from its peer. Initially, these two generation numbers are set to zero.

A host MAY increment its local generation number beyond the remote generation number it has recorded. We call this action _initiating re-keying_.

When a host has incremented its local generation number and uses the new key-set for the first time to encrypt an outgoing frame, it MUST set "rekey = 1" for that frame. It MUST set "rekey = 0" in all other cases.

When a host receives a frame with "rekey = 1", it increments its record of the remote generation number. If the remote generation number is now greater than the local generation number, the receiver MUST immediately increment its local generation number to match. Moreover, if the receiver has not yet transmitted a segment with the

FIN flag set, it MUST immediately send a frame (with empty application data if necessary) with "rekey = 1".

A host MUST NOT initiate more than one concurrent re-key operation if it has no data to send; that is, it MUST NOT initiate re-keying with an empty encryption frame more than once while its record of the remote generation number is less than its own.

Note that when parts of the datastream are retransmitted, TCP requires that implementations always send the same data bytes for the same TCP sequence numbers. Thus, frame data in retransmitted segments MUST be encrypted with the same key as when it was first transmitted, regardless of the current local generation number.

Implementations SHOULD delete older-generation keys from memory once they have received all frames they will need to decrypt with the old keys and have encrypted all outgoing frames under the old keys.

3.9. Keep-Alive

Instead of using TCP Keep-Alives to verify that the remote endpoint is still responsive, tcpcrypt implementations SHOULD employ the re-keying mechanism for this purpose, as follows. When necessary, a host SHOULD probe the liveness of its peer by initiating re-keying and transmitting a new frame immediately (with empty application data if necessary).

As described in Section 3.8, a host receiving a frame encrypted under a generation number greater than its own MUST increment its own generation number and (if it has not already transmitted a segment with FIN set) immediately transmit a new frame (with zero-length application data if necessary).

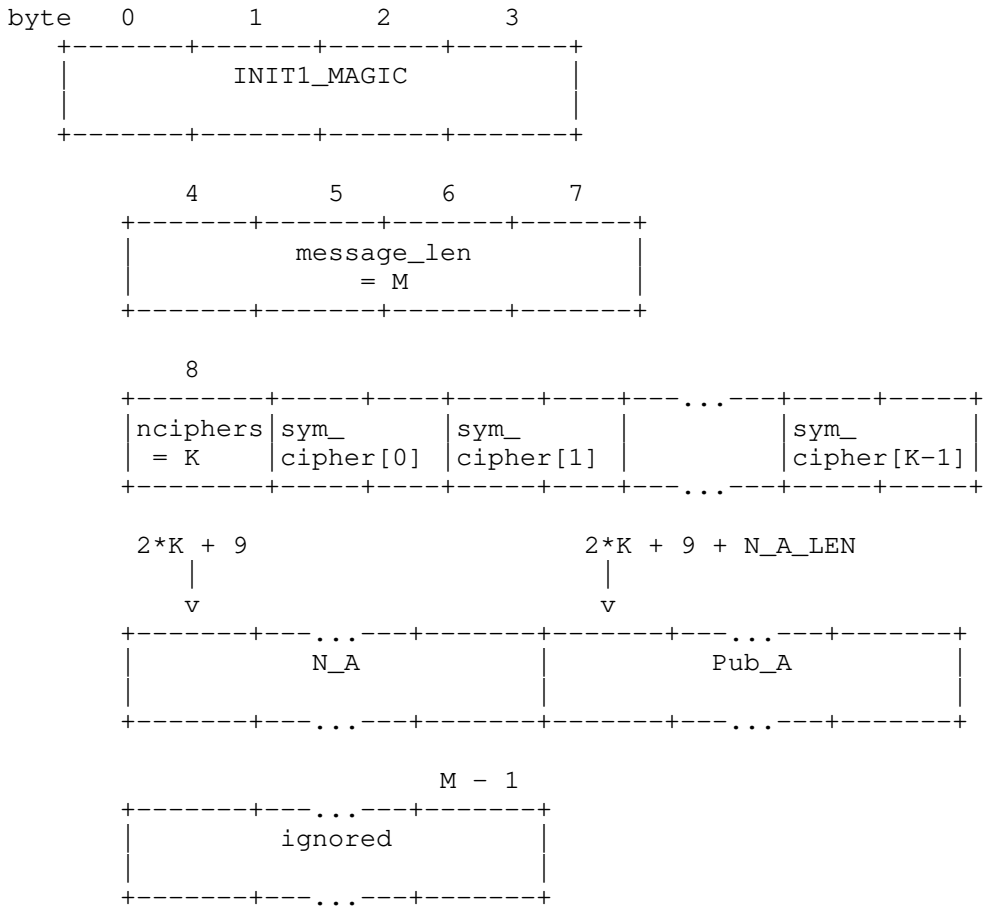
Implementations MAY use TCP Keep-Alives for purposes that do not require endpoint authentication, as discussed in Section 8.2.

4. Encodings

This section provides byte-level encodings for values transmitted or computed by the protocol.

4.1. Key-Exchange Messages

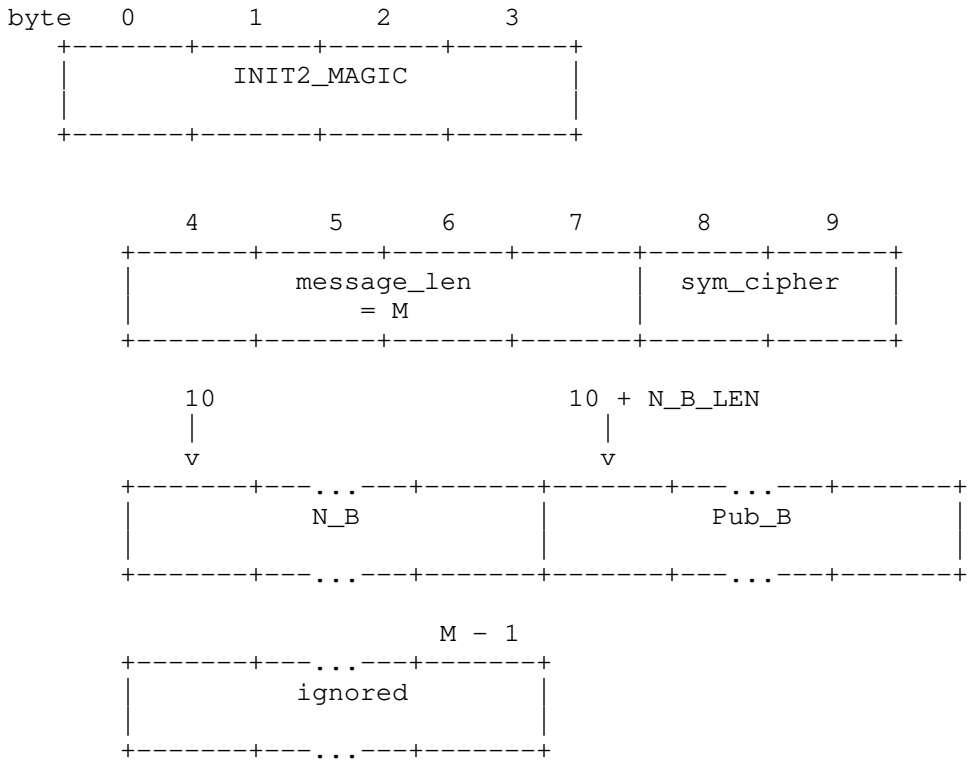
The "Init1" message has the following encoding:



The constant "INIT1_MAGIC" is defined in Section 4.3. The four-byte field "message_len" gives the length of the entire "Init1" message, encoded as a big-endian integer. The "nciphers" field contains an integer value that specifies the number of two-byte symmetric-cipher identifiers that follow. The "sym_cipher[i]" identifiers indicate cryptographic algorithms in Table 5 in Section 7. The length "N_A_LEN" and the length of "Pub_A" are both determined by the negotiated TEP, as described in Section 5.

Implementations of this protocol MUST construct "Init1" such that the field "ignored" has zero length; that is, they MUST construct the message such that its end, as determined by "message_len", coincides with the end of the field "Pub_A". When receiving "Init1", however, implementations MUST permit and ignore any bytes following "Pub_A".

The "Init2" message has the following encoding:

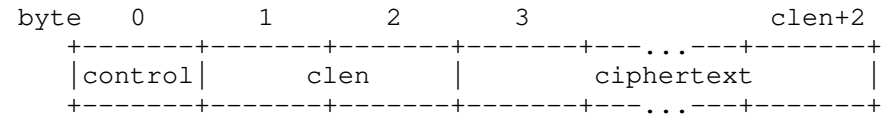


The constant "INIT2_MAGIC" is defined in Section 4.3. The four-byte field "message_len" gives the length of the entire "Init2" message, encoded as a big-endian integer. The "sym_cipher" value is a selection from the symmetric-cipher identifiers in the previously-received "Init1" message. The length "N_B_LEN" and the length of "Pub_B" are both determined by the negotiated TEP, as described in Section 5.

Implementations of this protocol MUST construct "Init2" such that the field "ignored" has zero length; that is, they MUST construct the message such that its end, as determined by "message_len", coincides with the end of the "Pub_B" field. When receiving "Init2", however, implementations MUST permit and ignore any bytes following "Pub_B".

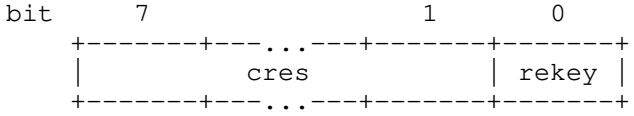
4.2. Encryption Frames

An _encryption frame_ comprises a control byte and a length-prefixed ciphertext value:



The field "clen" is an integer in big-endian format and gives the length of the "ciphertext" field.

The byte "control" has this structure:

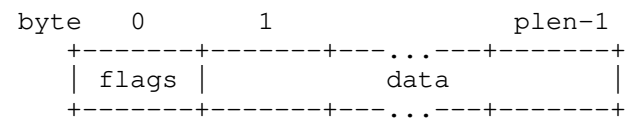


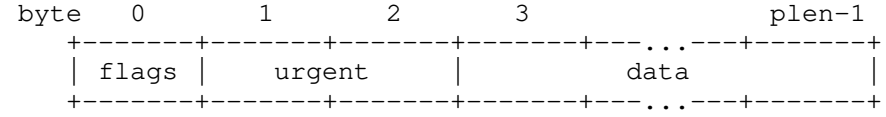
The seven-bit field "cres" is reserved; implementations MUST set these bits to zero when sending, and MUST ignore them when receiving.

The use of the "rekey" field is described in Section 3.8.

4.2.1. Plaintext

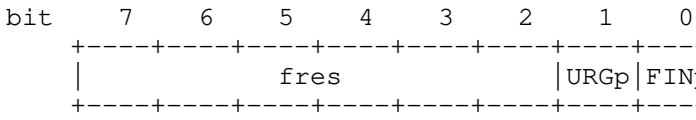
The "ciphertext" field is the result of applying the negotiated authenticated-encryption algorithm to a "plaintext" value, which has one of these two formats:





(Note that "clen" in the previous section will generally be greater than "plen", as the ciphertext produced by the authenticated-encryption scheme both encrypts the application data and provides redundancy with which to verify its integrity.)

The "flags" byte has this structure:



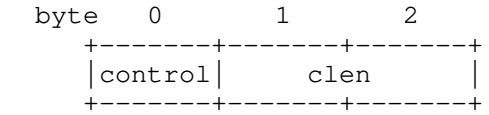
The six-bit value "fres" is reserved; implementations MUST set these six bits to zero when sending, and MUST ignore them when receiving.

When the "URGp" bit is set, it indicates that the "urgent" field is present, and thus that the plaintext value has the second structure variant above; otherwise the first variant is used.

The meaning of "urgent" and of the flag bits is described in Section 3.7.

4.2.2. Associated Data

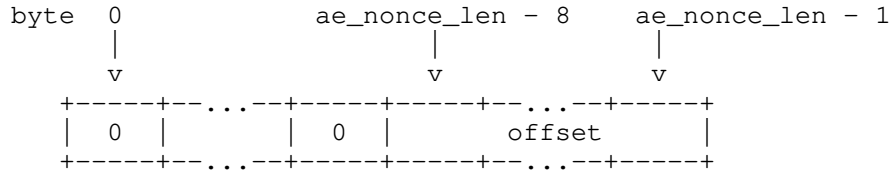
An encryption frame's "associated data" (which is supplied to the AEAD algorithm when decrypting the ciphertext and verifying the frame's integrity) has this format:



It contains the same values as the frame's "control" and "clen" fields.

4.2.3. Frame ID

Lastly, a "frame ID" (used to construct the nonce for the AEAD algorithm) has this format:



The 8-byte "offset" field contains an integer in big-endian format. Its value is specified in Section 3.6. Zero-valued bytes are prepended to the "offset" field to form a structure of length "ae_nonce_len".

4.3. Constant Values

The table below defines values for the constants used in the protocol.

Value	Name
0x01	CONST_NEXTK
0x02	CONST_SESSID
0x03	CONST_REKEY
0x04	CONST_KEY_A
0x05	CONST_KEY_B
0x06	CONST_RESUME
0x15101a0e	INIT1_MAGIC
0x097105e0	INIT2_MAGIC

Table 1: Constant values used in the protocol

5. Key-Agreement Schemes

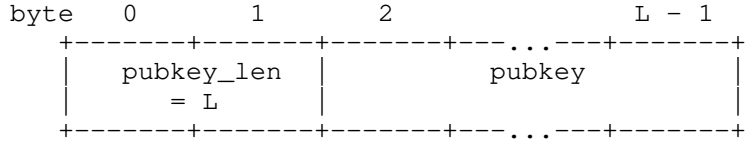
The TEP negotiated via TCP-ENO indicates the use of one of the key-agreement schemes named in Table 4 in Section 7. For example, "TCPCRYPT_ECDHE_P256" names the tcpcrypt protocol using ECDHE-P256 together with the CPRF and length parameters specified below.

All the TEPs specified in this document require the use of HKDF-Expand-SHA256 as the CPRF, and these lengths for nonces and session secrets:

```
N_A_LEN: 32 bytes
N_B_LEN: 32 bytes
K_LEN:   32 bytes
```

Future documents assigning additional TEPs for use with tcpcrypt mmight specify different values for the lengths above. Note that the minimum session ID length specified by TCP-ENO, together with the way tcpcrypt constructs session IDs, implies that "K_LEN" MUST have length at least 32 bytes.

Key-agreement schemes ECDHE-P256 and ECDHE-P521 employ the ECSVDP-DH secret value derivation primitive defined in [IEEE-1363]. The named curves are defined in [NIST-DSS]. When the public-key values "Pub_A" and "Pub_B" are transmitted as described in Section 4.1, they are encoded with the "Elliptic Curve Point to Octet String Conversion Primitive" described in Section E.2.3 of [IEEE-1363], and are prefixed by a two-byte length in big-endian format:



Implementations MUST encode these "pubkey" values in "compressed format". Implementations MUST validate these "pubkey" values according to the algorithm in [IEEE-1363] Section A.16.10.

Key-agreement schemes ECDHE-Curve25519 and ECDHE-Curve448 perform the Diffie-Helman protocol using the functions X25519 and X448, respectively. Implementations SHOULD compute these functions using the algorithms described in [RFC7748]. When they do so, implementations MUST check whether the computed Diffie-Hellman shared secret is the all-zero value and abort if so, as described in Section 6 of [RFC7748]. Alternative implementations of these functions SHOULD abort when either input forces the shared secret to one of a small set of values, as discussed in Section 7 of [RFC7748].

For these schemes, public-key values "Pub_A" and "Pub_B" are transmitted directly with no length prefix: 32 bytes for ECDHE-Curve25519, and 56 bytes for ECDHE-Curve448.

Table 2 below specifies the requirement levels of the four TEPs specified in this document. In particular, all implementations of tcpcrypt MUST support TCPCRYPT_ECDHE_Curve25519. However, system administrators MAY configure which TEPs a host will negotiate independent of these implementation requirements.

Requirement	TEP
REQUIRED	TCPCRYPT_ECDHE_Curve25519
RECOMMENDED	TCPCRYPT_ECDHE_Curve448
OPTIONAL	TCPCRYPT_ECDHE_P256
OPTIONAL	TCPCRYPT_ECDHE_P521

Table 2: Requirements for implementation of TEPs

6. AEAD Algorithms

This document uses "sym-cipher" identifiers in the messages "Init1" and "Init2" (see Section 3.3) to negotiate the use of AEAD algorithms; the values of these identifiers are given in Table 5 in Section 7. The algorithms "AEAD_AES_128_GCM" and "AEAD_AES_256_GCM"

are specified in [RFC5116]. The algorithm "AEAD_CHACHA20_POLY1305" is specified in [RFC7539].

Implementations MUST support certain AEAD algorithms according to Table 3 below. Note that system administrators MAY configure which algorithms a host will negotiate independent of these requirements.

Lastly, this document uses the lengths "ae_key_len" and "ae_nonce_len" to specify aspects of encryption and data formats. These values depend on the negotiated AEAD algorithm, also according to the table below.

AEAD Algorithm	Requirement	ae_key_len	ae_nonce_len
AEAD_AES_128_GCM	REQUIRED	16 bytes	12 bytes
AEAD_AES_256_GCM	RECOMMENDED	32 bytes	12 bytes
AEAD_CHACHA20_POLY1305	RECOMMENDED	32 bytes	12 bytes

Table 3: Requirement and lengths for each AEAD algorithm

7. IANA Considerations

For use with TCP-ENO's negotiation mechanism, tcpcrypt's TEP identifiers will need to be incorporated in IANA's "TCP encryption protocol identifiers" registry under the "Transmission Control Protocol (TCP) Parameters" registry, as in Table 4 below. The various key-agreement schemes used by these tcpcrypt variants are defined in Section 5.

Value	Meaning	Reference
0x21	TCPCRYPT_ECDHE_P256	[RFC-TBD]
0x22	TCPCRYPT_ECDHE_P521	[RFC-TBD]
0x23	TCPCRYPT_ECDHE_Curve25519	[RFC-TBD]
0x24	TCPCRYPT_ECDHE_Curve448	[RFC-TBD]

Table 4: TEP identifiers for use with tcpcrypt

In Section 6, this document defines the use of several AEAD algorithms for encrypting application data. To name these algorithms, the tcpcrypt protocol uses two-byte identifiers in the range 0x0001 to 0xFFFF inclusive, for which IANA is to maintain a new "tcpcrypt AEAD Algorithm" registry under the "Transmission Control Protocol (TCP) Parameters" registry. The initial values for this

registry are given in Table 5 below. Future assignments are to be made upon satisfying either of two policies defined in [RFC8126]: "IETF Review" or (for non-IETF stream specifications) "Expert Review with RFC Required." IANA will furthermore provide early allocation [RFC7120] to facilitate testing before RFCs are finalized.

Value	AEAD Algorithm	Reference
0x0001	AEAD_AES_128_GCM	[RFC-TBD] Section 6
0x0002	AEAD_AES_256_GCM	[RFC-TBD] Section 6
0x0010	AEAD_CHACHA20_POLY1305	[RFC-TBD] Section 6

Table 5: Authenticated-encryption algorithms for use with tcpcrypt

8. Security Considerations

All of the security considerations of TCP-ENO apply to tcpcrypt. In particular, tcpcrypt does not protect against active network attackers unless applications authenticate the session ID. If it can be established that the session IDs computed at each end of the connection match, then tcpcrypt guarantees that no man-in-the-middle attacks occurred unless the attacker has broken the underlying cryptographic primitives (e.g., ECDH). A proof of this property for an earlier version of the protocol has been published [tcpcrypt].

To ensure middlebox compatibility, tcpcrypt does not protect TCP headers. Hence, the protocol is vulnerable to denial-of-service from off-path attackers just as plain TCP is. Possible attacks include desynchronizing the underlying TCP stream, injecting RST or FIN segments, and forging re-key bits. These attacks will cause a tcpcrypt connection to hang or fail with an error, but not in any circumstance where plain TCP could continue uncorrupted. Implementations MUST give higher-level software a way to distinguish such errors from a clean end-of-stream (indicated by an authenticated "FINp" bit) so that applications can avoid semantic truncation attacks.

There is no "key confirmation" step in tcpcrypt. This is not needed because tcpcrypt's threat model includes the possibility of a connection to an adversary. If key negotiation is compromised and yields two different keys, failed integrity checks on every subsequent frame will cause the connection either to hang or to abort. This is not a new threat as an active attacker can achieve the same results against a plain TCP connection by injecting RST segments or modifying sequence and acknowledgement numbers.

Tcpcrypt uses short-lived public keys to provide forward secrecy. That is, once an implementation removes these keys from memory, a compromise of the system will not provide any means to derive the session secrets for past connections. All currently-specified key agreement schemes involve ECDHE-based key agreement, meaning a new key-pair can be efficiently computed for each connection. If implementations reuse these parameters, they MUST limit the lifetime of the private parameters as far as practical in order to minimize the number of past connections that are vulnerable. Of course, placing private keys in persistent storage introduces severe risks that they will not be destroyed reliably and in a timely fashion, and SHOULD be avoided whenever possible.

Attackers cannot force passive openers to move forward in their session resumption chain without guessing the content of the resumption identifier, which will be difficult without key knowledge.

The cipher-suites specified in this document all use HMAC-SHA256 to implement the collision-resistant pseudo-random function denoted by "CPRF". A collision-resistant function is one for which, for sufficiently large L, an attacker cannot find two distinct inputs (K_1, CONST_1) and (K_2, CONST_2) such that CPRF(K_1, CONST_1, L) = CPRF(K_2, CONST_2, L). Collision resistance is important to assure the uniqueness of session IDs, which are generated using the CPRF.

Lastly, many of tcpcrypt's cryptographic functions require random input, and thus any host implementing tcpcrypt MUST have access to a cryptographically-secure source of randomness or pseudo-randomness. [RFC4086] provides recommendations on how to achieve this.

Most implementations will rely on a device's pseudo-random generator, seeded from hardware events and a seed carried over from the previous boot. Once a pseudo-random generator has been properly seeded, it can generate effectively arbitrary amounts of pseudo-random data. However, until a pseudo-random generator has been seeded with sufficient entropy, not only will tcpcrypt be insecure, it will reveal information that further weakens the security of the pseudo-random generator, potentially harming other applications. As REQUIRED by TCP-ENO, implementations MUST NOT send ENO options unless they have access to an adequate source of randomness.

8.1. Asymmetric Roles

Tcpcrypt transforms a shared pseudo-random key (PRK) into cryptographic traffic keys for each direction. Doing so requires an asymmetry in the protocol, as the key derivation function must be perturbed differently to generate different keys in each direction. Tcpcrypt includes other asymmetries in the roles of the two hosts,

such as the process of negotiating algorithms (e.g., proposing vs. selecting cipher suites).

8.2. Verified Liveness

Many hosts implement TCP Keep-Alives [RFC1122] as an option for applications to ensure that the other end of a TCP connection still exists even when there is no data to be sent. A TCP Keep-Alive segment carries a sequence number one prior to the beginning of the send window, and may carry one byte of "garbage" data. Such a segment causes the remote side to send an acknowledgment.

Unfortunately, tcpcrypt cannot cryptographically verify Keep-Alive acknowledgments. Hence, an attacker could prolong the existence of a session at one host after the other end of the connection no longer exists. (Such an attack might prevent a process with sensitive data from exiting, giving an attacker more time to compromise a host and extract the sensitive data.)

To counter this threat, tcpcrypt specifies a way to stimulate the remote host to send verifiably fresh and authentic data, described in Section 3.9.

The TCP keep-alive mechanism has also been used for its effects on intermediate nodes in the network, such as preventing flow state from expiring at NAT boxes or firewalls. As these purposes do not require the authentication of endpoints, implementations MAY safely accomplish them using either the existing TCP keep-alive mechanism or tcpcrypt's verified keep-alive mechanism.

8.3. Mandatory Key-Agreement Schemes

This document mandates that tcpcrypt implementations provide support for at least one key-agreement scheme: ECDHE using Curve25519. This choice of a single mandatory algorithm is the result of a difficult tradeoff between cryptographic diversity and the ease and security of actual deployment.

The IETF's appraisal of best current practice on this matter [RFC7696] says, "Ideally, two independent sets of mandatory-to-implement algorithms will be specified, allowing for a primary suite and a secondary suite. This approach ensures that the secondary suite is widely deployed if a flaw is found in the primary one."

To meet that ideal, it might appear natural to also mandate ECDHE using P-256. However, implementing the Diffie-Hellman function using NIST elliptic curves (including those specified for use with tcpcrypt, P-256 and P-521) appears to be very difficult to achieve

without introducing vulnerability to side-channel attacks [NIST-fail]. Although well-trusted implementations are available as part of large cryptographic libraries, these can be difficult to extract for use in operating-system kernels where tcpcrypt is usually best implemented. In contrast, the characteristics of Curve25519 together with its recent popularity has led to many safe and efficient implementations, including some that fit naturally into the kernel environment.

[RFC7696] insists that, "The selected algorithms need to be resistant to side-channel attacks and also meet the performance, power, and code size requirements on a wide variety of platforms." On this principle, tcpcrypt excludes the NIST curves from the set of mandatory-to-implement key-agreement algorithms.

Lastly, this document encourages support for key-agreement with Curve448, categorizing it as RECOMMENDED. Curve448 appears likely to admit safe and efficient implementations. However, support is not REQUIRED because existing implementations might not yet be sufficiently well-proven.

9. Experiments

Some experience will be required to determine whether the tcpcrypt protocol can be deployed safely and successfully across the diverse environments of the global internet.

Safety means that TCP implementations that support tcpcrypt are able to communicate reliably in all the same settings as they would without tcpcrypt. As described in [I-D.ietf-tcpinc-tcpeno] Section 9, this property can be subverted if middleboxes strip ENO options from non-SYN segments after allowing them in SYN segments; or if the particular communication patterns of tcpcrypt offend the policies of middleboxes doing deep-packet inspection.

Success, in addition to safety, means hosts that implement tcpcrypt actually enable encryption when connecting to one another. This property depends on the network's treatment of the TCP-ENO handshake, and can be subverted if middleboxes merely strip unknown TCP options or if they terminate TCP connections and relay data back and forth unencrypted.

Ease of implementation will be a further challenge to deployment. Because tcpcrypt requires encryption operations on frames that may span TCP segments, kernel implementations are forced to buffer segments in different ways than are necessary for plain TCP. More implementation experience will show how much additional code

complexity is required in various operating systems, and what kind of performance effects can be expected.

10. Acknowledgments

We are grateful for contributions, help, discussions, and feedback from the TCPINC working group and from other IETF reviewers, including Marcelo Bagnulo, David Black, Bob Briscoe, Jana Iyengar, Stephen Kent, Tero Kivinen, Mirja Kuhlewind, Yoav Nir, Christoph Paasch, Eric Rescorla, Kyle Rose, and Dale Worley.

This work was funded by gifts from Intel (to Brad Karp) and from Google; by NSF award CNS-0716806 (A Clean-Slate Infrastructure for Information Flow Control); by DARPA CRASH under contract #N66001-10-2-4088; and by the Stanford Secure Internet of Things Project.

11. Contributors

Dan Boneh and Michael Hamburg were co-authors of the draft that became this document.

12. References

12.1. Normative References

[I-D.ietf-tcpinc-tcpeno]
Bittau, A., Giffin, D., Handley, M., Mazieres, D., and E. Smith, "TCP-ENO: Encryption Negotiation Option", draft-ietf-tcpinc-tcpeno-19 (work in progress), June 2018.

[IEEE-1363]
IEEE, "IEEE Standard Specifications for Public-Key Cryptography (IEEE Std 1363-2000)", 2000.

[NIST-DSS]
NIST, "FIPS PUB 186-4: Digital Signature Standard (DSS)", 2013.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[RFC2018] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, DOI 10.17487/RFC2018, October 1996, <https://www.rfc-editor.org/info/rfc2018>.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

[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>.

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

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

[RFC7120] Cotton, M., "Early IANA Allocation of Standards Track Code Points", BCP 100, RFC 7120, DOI 10.17487/RFC7120, January 2014, <https://www.rfc-editor.org/info/rfc7120>.

[RFC7539] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 7539, DOI 10.17487/RFC7539, May 2015, <https://www.rfc-editor.org/info/rfc7539>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

[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>.

12.2. Informative References

[I-D.ietf-tcpinc-api] Bittau, A., Boneh, D., Giffin, D., Handley, M., Mazieres, D., and E. Smith, "Interface Extensions for TCP-ENO and tcpcrypt", draft-ietf-tcpinc-api-06 (work in progress), June 2018.

[NIST-fail]
Bernstein, D. and T. Lange, "Failures in NIST's ECC standards", 2016, <https://cr.yp.to/newelliptic/nistecc-20160106.pdf>.

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <https://www.rfc-editor.org/info/rfc1122>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[RFC7696] Housley, R., "Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms", BCP 201, RFC 7696, DOI 10.17487/RFC7696, November 2015, <https://www.rfc-editor.org/info/rfc7696>.

[tcpcrypt]
Bittau, A., Hamburg, M., Handley, M., Mazieres, D., and D. Boneh, "The case for ubiquitous transport-level encryption", USENIX Security , 2010.

Authors' Addresses

Andrea Bittau
Google
345 Spear Street
San Francisco, CA 94105
US

Email: bittau@google.com

Daniel B. Giffin
Stanford University
353 Serra Mall, Room 288
Stanford, CA 94305
US

Email: dbg@scs.stanford.edu

Mark Handley
University College London
Gower St.
London WC1E 6BT
UK

Email: M.Handley@cs.ucl.ac.uk

David Mazieres
Stanford University
353 Serra Mall, Room 290
Stanford, CA 94305
US

Email: dm@uun.org

Quinn Slack
Sourcegraph
121 2nd St Ste 200
San Francisco, CA 94105
US

Email: sqs@sourcegraph.com

Eric W. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304
US

Email: eric.smith@kestrel.edu

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: April 2, 2016

A. Bittau
D. Boneh
D. Giffin
Stanford University
M. Handley
University College London
D. Mazieres
Stanford University
E. Smith
Kestrel Institute

September 30, 2015

TCP-ENO: Encryption Negotiation Option
draft-ietf-tcpinc-tcpeno-00

Abstract

Despite growing adoption of TLS [RFC5246], a significant fraction of TCP traffic on the Internet remains unencrypted. The persistence of unencrypted traffic can be attributed to at least two factors. First, some legacy protocols lack a signaling mechanism (such as a "STARTTLS" command) by which to convey support for encryption, making incremental deployment impossible. Second, legacy applications themselves cannot always be upgraded, requiring a way to implement encryption transparently entirely within the transport layer. The TCP Encryption Negotiation Option (TCP-ENO) addresses both of these problems through a new TCP option kind providing out-of-band, fully backward-compatible negotiation of encryption.

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 April 2, 2016.

Table of Contents

1. Requirements language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Introduction

Many applications and protocols running on top of TCP today do not encrypt traffic. This failure to encrypt lowers the bar for certain attacks, harming both user privacy and system security. Counteracting the problem demands a minimally intrusive, backward-compatible mechanism for incrementally deploying encryption. The TCP Encryption Negotiation Option (TCP-ENO) specified in this document provides such a mechanism.

While the need for encryption is immediate, future developments could alter trade-offs and change the best approach to TCP-level encryption (beyond introducing new cipher suites). For example:

- Increased option space in TCP [I-D.ietf-tcpm-tcp-edo][I-D.briscoe-tcpm-inspace-mode-tcpbis][I-D.touch-tcpm-tcp-syn-ext-opt] could reduce round trip times and simplify protocols.

- API revisions to socket interfaces [RFC3493] could benefit from integration with TCP-level encryption, particularly if combined with technologies such as DANE [RFC6394].

- The forthcoming TLS 1.3 [I-D.ietf-tls-tls13] standard could reach more applications given an out-of-band, backward-compatible mechanism for enabling encryption.

- TCP fast open [RFC7413], as it gains more widespread adoption and middlebox acceptance, could potentially benefit from tailored encryption support.

- Cryptographic developments that either shorten or lengthen the minimal key exchange messages required could affect how such messages are best encoded in TCP segments.

Introducing TCP options, extending operating system interfaces to support TCP-level encryption, and extending applications to take advantage of TCP-level encryption will all require effort. To the greatest extent possible, this effort ought to remain applicable if the need arises to change encryption strategies. To this end, it is useful to consider two questions separately:

1. How to negotiate the use of encryption at the TCP layer, and

2. How to perform encryption at the TCP layer.

This document addresses question 1 with a new option called TCP-ENO. TCP-ENO provides a framework in which two endpoints can agree on one among multiple possible TCP encryption _specs_. For future

compatibility, encryption specs can vary widely in terms of wire format, use of TCP option space, and integration with the TCP header and segmentation. A companion document, the TCPINC encryption spec, addresses question 2. TCPINC enables TCP-level traffic encryption today. TCP-ENO ensures that the effort invested to deploy TCPINC can benefit future encryption specs should a different approach at some point be preferable.

At a lower level, TCP-ENO was designed to achieve the following goals:

1. Enable endpoints to negotiate the use of a separately specified encryption _spec_.

2. Transparently fall back to unencrypted TCP when not supported by both endpoints.

3. Provide signaling through which applications can better take advantage of TCP-level encryption (for instance by improving authentication mechanisms in the presence of TCP-level encryption).

4. Provide a standard negotiation transcript through which specs can defend against tampering with TCP-ENO.

5. Make parsimonious use of TCP option space.

6. Define roles for the two ends of a TCP connection, so as to name each end of a connection for encryption or authentication purposes even following a symmetric simultaneous open.

3. The TCP-ENO option

TCP-ENO is a TCP option used during connection establishment to negotiate how to encrypt traffic. As an option, TCP-ENO can be deployed incrementally. Legacy hosts unaware of the option simply ignore it and never send it, causing traffic to fall back to unencrypted TCP. Similarly, middleboxes that strip out unknown options including TCP-ENO will downgrade connections to plaintext without breaking them. Of course, downgrading makes TCP-ENO vulnerable to active attackers, but appropriately modified applications can protect themselves by considering the state of TCP-level encryption during authentication, as discussed in Section 7.

The ENO option takes two forms. In TCP segments with the SYN flag set, it acts as a container for a series of one or more suboptions, labeled "Opt_0", "Opt_1", ... in Figure 1. In non-SYN segments, ENO conveys only a single bit of information, namely an acknowledgment

that the sender received an ENO option in the other host's SYN segment. (Such acknowledgments enable graceful fallback to unencrypted TCP in the event that a middlebox strips ENO options in one direction.) Figure 2 illustrates the non-SYN form of the ENO option. Encryption specs MAY include extra bytes in a non-SYN ENO option, but TCP-ENO itself MUST ignore them. In accordance with TCP [RFC0793], the first two bytes of the ENO option always consist of the kind (ENO) and the total length of the option.

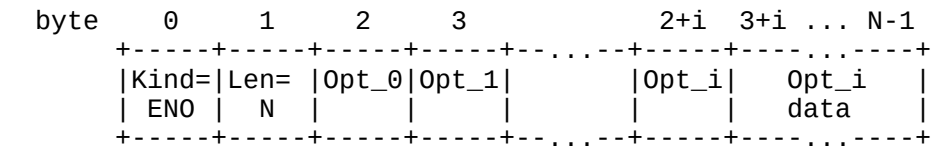


Figure 1: TCP-ENO option in SYN segment (MUST contain at least one suboption)

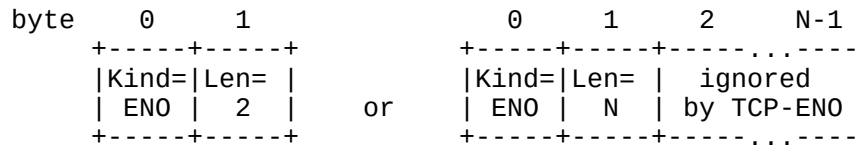
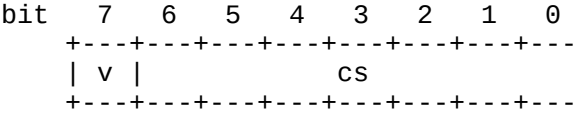


Figure 2: non-SYN TCP-ENO option in segment without SYN flag

Every suboption starts with a byte of the form illustrated in Figure 3. The seven-bit value "cs" specifies the meaning of the suboption. Each value of "cs" either specifies general parameters (discussed in Section 3.3) or indicates the willingness to use a specific encryption spec detailed in a separate document.



v - 1 when suboption followed by variable-length data
cs - global configuration option or encryption spec identifier

Figure 3: Format of suboption byte

The high bit "v" in a suboption's first byte specifies whether or not the suboption is followed by variable-length data. If "v" is 0, the suboption consists of only the one byte shown in Figure 3. If "v" is 1, then the suboption is followed by variable-length data. Suboption data MAY be used for session caching, cipher suite negotiation, key exchange, or other purposes, as determined by the value of "cs".

Every suboption but the last in an ENO option MUST be a one-byte suboption (with "v" = 0). The last suboption MAY be a variable-length suboption. Its length is determined by the total length of the TCP option. In Figure 1, "Opt_i" is the variable-length option; its total size is N-(2+i) bytes--one byte for "Opt_i" itself and N-(3+i) bytes for additional data. Multiple suboptions with data may be included in a single TCP SYN segment by repeating the ENO option.

Table 1 summarizes the allocation of values of "cs". Values under 0x10 are assigned to _general suboptions_ whose meaning applies across encryption specs, as discussed in Section 3.3. Values greater than or equal to 0x20 will be assigned to _spec identifiers_. Values in the range 0x10-0x1f are reserved for possible future general options. Implementations MUST ignore all unknown suboptions.

cs	Meaning
0x00-0x0f	General options (see Section 3.3)
0x10-0x1f	Reserved for possible use by future general options
0x20-0x7f	Used to designate encryption specs

Table 1: Allocation of cs bits in TCP-ENO suboptions

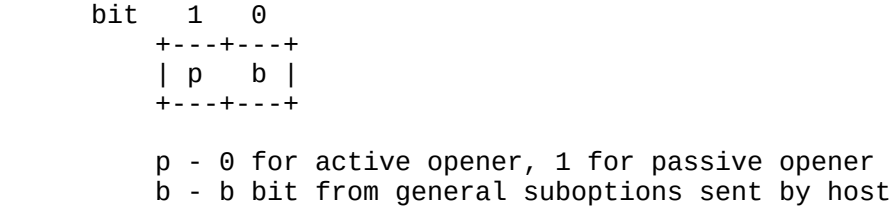
3.1. TCP-ENO roles

TCP-ENO uses abstract roles to distinguish the two ends of a TCP connection: One host plays the "A" role, while the other host plays the "B" role. Following a normal three-way handshake, the active opener plays the A role and the passive opener plays the B role. An active opener is a host that sends a SYN segment without the ACK flag set (after a "connect" system call on socket-based systems). A passive opener's SYN segment always contains the ACK flag (and follows a "listen" call on socket-based systems).

Roles are abstracted from the active/passive opener distinction to deal with simultaneous open, in which both hosts are active openers. For simultaneous open, the general suboptions discussed in Section 3.3 define a tie-breaker bit "b", where the host with "b = 1" plays the B role, and the host with "b = 0" plays the A role. If two active openers have the same "b" bit, TCP-ENO fails and reverts to unencrypted TCP.

More precisely, the above role assignment can be reduced to comparing a two-bit role _priority_ for each host, shown in Figure 4. The most significant bit, "p", is 1 for a passive opener and 0 for an active opener. The least-significant bit "b" is the tie-breaker bit. The

host with the lower priority assumes the A role; the host with the higher priority assumes the B role. In the event of a tie, TCP-ENO fails and MUST continue with unencrypted TCP as if the ENO options had not been present in SYN segments.



p - 0 for active opener, 1 for passive opener
b - b bit from general suboptions sent by host

Figure 4: Role priority of an endpoint

Encryption specs SHOULD refer to TCP-ENO's A and B roles to specify asymmetric behavior by the two hosts. For the remainder of this document, we will use the terms "host A" and "host B" to designate the hosts with role A and B respectively in a connection.

3.2. TCP-ENO handshake

The TCP-ENO option is intended for use during TCP connection establishment. To enable incremental deployment, a host needs to ensure both that the other host supports TCP-ENO and that no middlebox has stripped the ENO option from its own TCP segments. In the event that either of these conditions does not hold, implementations MUST immediately cease sending TCP-ENO options and MUST continue with unencrypted TCP as if the ENO option had not been present.

More precisely, for negotiation to succeed, the TCP-ENO option MUST be present in the SYN segment sent by each host, so as to indicate support for TCP-ENO. Additionally, the ENO option MUST be present in the first ACK segment sent by each host, so as to indicate that no middlebox stripped the ENO option from the ACKed SYN. Depending on whether a host is an active or a passive opener, the first ACK segment may or may not be the same as the SYN segment. Specifically:

- An active opener begins with a SYN-only segment, and hence must send two segments containing ENO options. The initial SYN-only segment MUST contain an ENO option with at least one suboption, as pictured in Figure 1. If ENO succeeds, the active opener's first ACK segment MUST subsequently contain a non-SYN ENO option, as pictured in Figure 2.

- A passive opener's first transmitted segment has both the SYN and ACK flags set. Therefore, a passive opener sends an ENO option of

the type shown in Figure 1 in its single SYN-ACK segment and does not send a non-SYN ENO option.

A spec identifier in one host's SYN segment is _valid_ if it is compatible with a suboption in the other host's SYN segment. Two suboptions are _compatible_ when they have the same "cs" value (>= 0x20) and when the particular combination of "v" bits and suboption data in suboptions of the two SYN segments is well-defined by the corresponding encryption spec. Specs MAY allow or disallow any combination of values of "v" in the two SYN segments.

Once the two sides have exchanged SYN segments, the _negotiated spec_ is the last valid spec identifier in the SYN segment of host B (that is, the passive opener in the absence of simultaneous open). In other words, the order of suboptions in host B's SYN segment determines spec priority, while the order of suboptions in host A's SYN segment has no effect. Hosts must disable TCP-ENO if there is no valid spec in host B's SYN segment. Note that negotiation prioritizes the last rather than the first valid suboption so as to favor the spec with suboption data, if there is one.

When possible, host B SHOULD send only one spec identifier (suboption in the range 0x20-0xff), and SHOULD ensure this option is valid. However, sending a single valid spec identifier is not required, as doing so could be impractical in some cases, such as simultaneous open or library-level implementations that can only provide a static TCP-ENO option to the kernel.

A host MUST disable ENO if any of the following conditions holds:

1. The host receives a SYN segment without an ENO option,

2. The host receives a SYN segment that contains no valid encryption specs when paired with the SYN segment that the host has already sent or would otherwise have sent,

3. The host receives a SYN segment containing general suboptions that are incompatible with the SYN segment that it has already sent or would otherwise have sent, or

4. The first ACK segment received by a host does not contain an ENO option.

After disabling ENO, a host MUST NOT transmit any further ENO options and MUST fall back to unencrypted TCP.

Conversely, if a host receives an ACK segment containing an ENO option, then encryption MUST be enabled. From this point the host

MUST follow the encryption protocol of the negotiated spec and MUST NOT present raw TCP payload data to the application. In particular, data segments MUST contain ciphertext or key agreement messages as determined by the negotiated spec, and MUST NOT contain plaintext application data.

3.2.1. Handshake examples

```
(1) A -> B:  SYN      ENO<X,Y>
(2) B -> A:  SYN-ACK  ENO<Y>
(3) A -> B:  ACK      ENO<>
[rest of connection encrypted according to spec for Y]
```

Figure 5: Three-way handshake with successful TCP-ENO negotiation

Figure 5 shows a three-way handshake with a successful TCP-ENO negotiation. The two sides agree to follow the encryption spec identified by suboption Y.

```
(1) A -> B:  SYN      ENO<X,Y>
(2) B -> A:  SYN-ACK
(3) A -> B:  ACK
[rest of connection unencrypted legacy TCP]
```

Figure 6: Three-way handshake with failed TCP-ENO negotiation

Figure 6 shows a failed TCP-ENO negotiation. The active opener (A) indicates support for specs corresponding to suboptions X and Y. Unfortunately, at this point one of thee things occurs:

1. The passive opener (B) does not support TCP-ENO,

2. B supports TCP-ENO, but supports neither of specs X and Y, and so does not reply with an ENO option, or

3. The network stripped the ENO option out of A’s SYN segment, so B did not receive it.

Whichever of the above applies, the connection transparently falls back to unencrypted TCP.

```
(1) A -> B:  SYN      ENO<X,Y>
(2) B -> A:  SYN-ACK  ENO<X>    [ENO stripped by middlebox]
(3) A -> B:  ACK
[rest of connection unencrypted legacy TCP]
```

Figure 7: Failed TCP-ENO negotiation because of network filtering

Figure 7 Shows another handshake with a failed encryption negotiation. In this case, the passive opener B receives an ENO option from A and replies. However, the reverse network path from B to A strips ENO options. Hence, A does not receive an ENO option from B, disables ENO, and does not include the required non-SYN ENO option when ACKing the other host's SYN segment. The lack of ENO in A's ACK segment signals to B that the connection will not be encrypted. At this point, the two hosts proceed with an unencrypted TCP connection.

```
(1) A -> B:  SYN      ENO<Y,X>
(2) B -> A:  SYN      ENO<0x01,X,Y,Z>
(3) A -> B:  SYN-ACK  ENO<Y,X>
(4) B -> A:  SYN-ACK  ENO<0x01,X,Y,Z>
[rest of connection encrypted according to spec for Y]
```

Figure 8: Simultaneous open with successful TCP-ENO negotiation

Figure 8 shows a successful TCP-ENO negotiation with simultaneous open. Here the first four segments MUST contain an ENO option, as each side sends both a SYN-only and a SYN-ACK segment. The ENO option in each hosts's SYN-ACK is identical to the ENO option in its SYN-only segment, as otherwise connection establishment could not recover from the loss of a SYN segment. Note the use of the tie-breaker bit in general suboption 0x01 assigns B its role, as discussed in Section 3.3. The last valid spec in B's ENO option is Y, so Y is the negotiated spec.

3.3. General suboptions

Suboptions 0x00-0x0f are used for general conditions that apply regardless of the negotiated encryption spec. A TCP segment MUST include at most one ENO suboption whose high nibble is 0. The value of the low nibble is interpreted as a bitmask, illustrated in Figure 9.

```
bit   7   6   5   4   3   2   1   0
    +---+---+---+---+---+-------+---+
    | 0   0   0   0   z    aa     b |
    +---+---+---+---+---+-------+---+

    z  - Zero bit (reserved for future use)
    aa - Application-aware bits
    b  - Tie-breaker bit for simultaneous open
```

Figure 9: Format of the general option byte

The fields of the bitmask are interpreted as follows:

z The "z" bit is reserved for future revisions of TCP-ENO. Its value MUST be set to zero in sent segments and ignored in received segments.

aa The two application-aware bits indicate that the application on the sending host is aware of TCP-ENO and has been extended to alter its behavior in the presence of encrypted TCP. There are four possible values, as shown in Table 2. The default, when applications have not been modified to take advantage of TCP-ENO, MUST be 00. However, implementations SHOULD provide an API through which applications can set the bits to other values and query for the other host's application-aware bits. The value 01 indicates that the application is aware of TCP-ENO. The value 10 (binary) is reserved for future use. It MUST be interpreted as the application being aware of TCP-ENO, but MUST never be sent.

Value 11 (binary) indicates that an application is aware of TCP-ENO and requires application awareness from the other side. If one host sends value 00 and the other host sends 11, then TCP-ENO MUST be disabled and fall back to unencrypted TCP. Any other combination of values (including the reserved 10) is compatible with enabling encryption. A possible use of value 11 is for applications that perform legacy encryption and wish to disable TCP-ENO unless higher-layer encryption can be disabled.

Value	Meaning
00	Application is not aware of TCP-ENO
01	Application is aware of TCP-ENO
10	Reserved but interpreted as ENO-aware
11	Application awareness is mandatory for use of TCP-ENO

Table 2: Meaning of the two application-aware bits

b This is the tie-breaker bit in role priority, discussed in Section 3.1.

A SYN segment without an explicit general suboption has an implicit general suboption of 0x00.

3.4. Negotiation transcript

To defend against attacks on encryption negotiation itself, encryption specs need a way to reference a transcript of TCP-ENO's negotiation. In particular, an encryption spec MUST fail with high

probability if its selection resulted from tampering with or forging initial SYN segments.

TCP-ENO defines its negotiation transcript as a packed data structure consisting of a series of TCP-ENO options (each including the ENO and length bytes, as they appeared in the TCP header). Specifically, the transcript is constructed from the following, in order:

1. Every TCP-ENO option in host A's SYN segment, including the kind and length bytes, in the order the options appeared in that SYN segment.

2. A minimal two-byte ENO option, as shown on the left in Figure 2.

3. Every TCP-ENO option in host B's SYN segment, including the kind and length bytes, in the order the options appeared in that SYN segment.

4. A minimal two-byte ENO option, as shown on the left in Figure 2.

Note that 2 and 4 merely serve as delimiters to separate the two hosts' options from each other and from any data that follows the transcript. Note further that any ignored data in non-SYN ENO options does not appear in the transcript. Because parts 2 and 4 are always exactly two bytes and SYN segments MUST NOT contain two-byte ENO options, this encoding is unambiguous.

For the transcript to be well defined, hosts MUST NOT alter ENO options in retransmitted segments, or between the SYN and SYN-ACK segments of a simultaneous open, except that an active opener MAY remove the ENO option altogether from a retransmitted SYN-only segment and disable TCP-ENO. Such removal could be useful if middleboxes are dropping segments with the ENO option.

4. Requirements for encryption specs

TCP-ENO was designed to afford encryption spec authors a large amount of design flexibility. Nonetheless, to fit all encryption specs into a coherent framework and abstract most of the differences away for application writers, all encryption specs claiming ENO "cs" numbers MUST satisfy the following properties.

- Specs MUST protect TCP data streams with authenticated encryption.

- Specs MUST define a session ID whose value identifies the TCP connection and, with overwhelming probability, is unique over all time if either host correctly obeys the spec. Section 4.1 describes the requirements of the session ID in more detail.

- Specs MUST NOT permit the negotiation of any encryption algorithms with significantly less than 128-bit security.

- Specs MUST NOT allow the negotiation of null cipher suites, even for debugging purposes. (Implementations MAY support debugging modes that allow applications to extract their own session keys.)

- Specs MUST NOT allow the negotiation of encryption modes that do not provide forward secrecy some bounded, short time after the close of a TCP connection.

- Specs MUST protect and authenticate the end-of-file marker traditionally conveyed by TCP's FIN flag when the remote application calls "close" or "shutdown". However, end-of-file MAY be conveyed though a mechanism other than TCP FIN. Moreover, specs MAY permit attacks that cause TCP connections to abort, but such an abort MUST raise an error that is distinct from an end-of-file condition.

- Specs MAY disallow the use of TCP urgent data by applications, but MUST NOT allow attackers to manipulate the URG flag and urgent pointer in ways that are visible to applications.

4.1. Session IDs

Each spec MUST define a session ID that uniquely identifies each encrypted TCP connection. Implementations SHOULD expose the session ID to applications via an API extension. Applications that are aware of TCP-ENO SHOULD incorporate the session ID value and TCP-ENO role (A or B) into any authentication mechanisms layered over TCP encryption so as to authenticate actual TCP endpoints.

In order to avoid replay attacks and prevent authenticated session IDs from being used out of context, session IDs MUST be unique over all time with high probability. This uniqueness property MUST hold even if one end of a connection maliciously manipulates the protocol in an effort to create duplicate session IDs. In other words, it MUST be infeasible for a host, even by deviating from the encryption spec, to establish two TCP connections with the same session ID to remote hosts obeying the spec.

To prevent session IDs from being confused across specs, all session IDs begin with the negotiated spec identifier--that is, the last valid spec identifier in host B's SYN segment. If the "v" bit was 1 in host B's SYN segment, then it is also 1 in the session ID. However, only the first byte is included, not the suboption data. Figure 10 shows the resulting format. This format is designed for spec authors to compute unique identifiers; it is not intended for

applications authors to pick apart session IDs. Applications SHOULD treat session IDs as monolithic opaque values and SHOULD NOT discard the first byte to shorten identifiers.

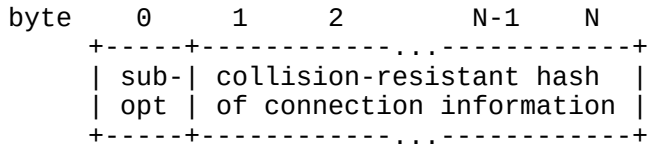


Figure 10: Format of a session ID

Though specs retain considerable flexibility in their definitions of the session ID, all session IDs MUST meet certain minimum requirements. In particular:

- The session ID MUST be at least 33 bytes (including the one-byte suboption), though specs may choose longer session IDs.
- The session ID MUST depend in a collision-resistant way on fresh data contributed by both sides of the connection.
- The session ID MUST depend in a collision-resistant way on any public keys, public Diffie-Hellman parameters, or other public asymmetric cryptographic parameters that are employed by the encryption spec and have corresponding private data that is known by only one side of the connection.
- Unless and until applications disclose information about the session ID, all but the first byte MUST be computationally indistinguishable from random bytes to a network eavesdropper.
- Applications MAY chose to make session IDs public. Therefore, specs MUST NOT place any confidential data in the session ID (such as data permitting the derivation of session keys).
- The session ID MUST depend on the negotiation transcript specified in Section 3.4 in a collision-resistant way.

4.2. Option kind sharing

This draft does not specify the use of ENO options in any segments other than the initial SYN and ACK segments of a connection. Moreover, it does not specify the content of ENO options in an initial ACK segment that has the SYN flag clear. As a result, any use of the ENO option kind after the SYN exchange will not conflict with TCP-ENO. Therefore, encryption specs that require TCP option

space MAY re-purpose the ENO option kind for use in segments after the initial SYN.

5. API extensions

Implementations SHOULD provide API extensions through which applications can query and configure the behavior of TCP-ENO, including retrieving session IDs, setting and reading application-aware bits, and specifying which specs to negotiate. The specifics of such an API are outside the scope of this document.

6. Open issues

This document has experimental status because of several open issues. Some questions about TCP-ENO's viability depend on middlebox behavior that can only be determined a posteriori. Hence, initial deployment of ENO will be an experiment. In addition, a few design questions exists on which consensus is not clear, and hence for which greater discussion and justification of TCP-ENO's design may be helpful.

6.1. Experiments

One of the primary open questions is to what extent middleboxes will permit the use of TCP-ENO. Once TCP-ENO is deployed, we will be in a better position to gather data on two types of failure:

1. Middleboxes downgrading TCP-ENO connections to unencrypted TCP. This can happen if middleboxes strip unknown TCP options or if they terminate TCP connections and relay data back and forth.

2. Middleboxes causing TCP-ENO connections to fail completely. This can happen if applications perform deep packet inspection and start dropping segments that unexpectedly contain ciphertext.

The first type of failure is tolerable since TCP-ENO is designed for incremental deployment anyway. The second type of failure is more problematic, and, if prevalent, will require the development of techniques to avoid and recover from such failures.

6.2. Simultaneous open

Simultaneous open is the only way to establish a TCP connection between TCP hosts in certain NAT configurations [RFC5382]. The principle challenge in simultaneous open is breaking TCP's symmetry for both sides to agree on the assignment of the A and B roles. Relying on TCP/IP header fields such as the IP address, port number, and initial sequence number is problematic as these values may be

modified by middleboxes, meaning a sender does not know what values the recipient will see for these fields.

The authors lack data on how prevalent simultaneous open is in the wild. The use of simultaneous open has been specified for ICE [RFC6544], but the highest profile implementation (the firefox browser) currently prefers UDP over TCP when permitted by firewalls. Moreover, applications of ICE typically already encrypt data and would disable TCP-ENO to avoid double encryption. It is therefore unclear what level of support TCP-ENO should provide for simultaneous open, or at what cost such support is justified. The working group has discussed four levels of support with no clear consensus:

1. Require applications to break the tie out of band and assign themselves A and B roles. If applications do not assign the roles properly, the TCP connection fails entirely.

2. As above, require applications to specify roles, but if they do so incorrectly fall back to unencrypted TCP.

3. Require applications to declare that they are using simultaneous open, but do not require them to negotiate roles. Leave it to TCP-ENO break the tie and negotiate roles.

4. Design TCP-ENO so that it works completely transparently in conjunction with simultaneous open, with no application involvement required.

This simplest and cheapest solution is obviously #1. This document currently embraces design point #2, at the cost of an extra bit (the "b" bit in the general suboption) for hosts to check whether roles were properly assigned. Solution #3 would likely consume 4-8 additional bytes of option space in the case of a simultaneous open, so as to include a random tie-breaker value. Solution #4 would consume 4-8 additional bytes of option space in every SYN segment, as current APIs make it impossible to distinguish a "connect" call intended for a simultaneous open from one intended for a three-way handshake.

6.3. Multiple Session IDs

Though currently specs must output a single session ID, it might alternatively be useful to define multiple identifiers per connection. As an example, a public session ID might be used to authenticate a connection, while a private session ID could be used as an authentication key to link out-of-band data (such as another TCP connection) to the original connection.

6.4. Suboption data

TCP-ENO currently optimizes for the case that a single suboption per SYN segment contains suboption data. This design was chosen in expectation that the following two use cases will be the most common:

- An active opener advertises support for multiple specs using one-byte suboptions. The passive opener picks one of the advertised specs and replies with a single suboption, possibly using suboption data for options within the negotiated spec. Such spec-specific options might convey supported elliptic curves or public key ciphers.

- An active opener advertises support for multiple specs as above, but also includes a single longer suboption containing a session caching cookie with which the hosts may be able to avoid the cost of public key cryptography. In this case, the server either accepts the cookie or reverts to picking one of the other specs as in the previous case.

Both of these use cases require at most one multi-byte suboption per SYN segment. To optimize for this case, TCP-ENO relies on the TCP option length byte to specify the length of the multi-byte suboption implicitly. Segments with more than one multi-byte suboption must repeat the ENO kind byte, losing one byte of precious TCP SYN option space.

An alternative would be for each multi-byte suboption to be followed by its own length field. This would cost an extra byte of SYN option space in the two cases above, but save one byte for each additional multi-byte suboption.

As an example, in the current ENO design, a SYN segment with ENO suboption containing 2 bytes of data consumes 5 bytes (the ENO kind, the TCP option length, the spec identifier, and 2 bytes of suboption data). An ENO option with two 2-byte suboptions requires double this, or 10 bytes. By contrast, in a design with a suboption length byte, one 2-byte suboption would cost 6 bytes (ENO kind, TCP option length, suboption, suboption length, and 2 bytes of option data), but two 2-byte suboptions could be packed together, without repeating the ENO kind byte, in only 9 bytes of option space.

In the event that the above two use cases are not the most prevalent, it may be worth revisiting ENO’s choice of optimized case.

7. Security considerations

An obvious use case for TCP-ENO is opportunistic encryption. However, if applications do not check and verify the session ID, they will be open to man-in-the-middle attacks as well as simple downgrade attacks in which an attacker strips off the TCP-ENO option. Hence, where possible, applications SHOULD be modified to fold the session ID into authentication mechanisms, and SHOULD employ the application-aware bits as needed to enable such negotiation in a backward-compatible way.

Because TCP-ENO enables multiple different encryption specs to coexist, security could potentially be only as strong as the weakest available encryption spec. For this reason, it is crucial for session IDs to depend on the TCP-ENO transcript in a strong way. Hence, encryption specs SHOULD compute session IDs using only well-studied and conservative hash functions. Thus, even if an encryption spec is broken, and even if people deprecate it instead of disabling it, and even if an attacker tampers with ENO options to force negotiation of the broken spec, it should still be intractable for the attacker to induce identical session IDs at both hosts.

Implementations MUST not send ENO options unless encryption specs have access to a strong source of randomness or pseudo-randomness. Without secret unpredictable data at both ends of a connection, it is impossible for encryption specs to satisfy the confidentiality and forward secrecy properties required by this document.

8. IANA Considerations

A new TCP option kind number needs to be assigned to ENO by IANA.

In addition, IANA will need to maintain an ENO suboption registry mapping suboption "cs" values to encryption specs.

9. Acknowledgments

This work was funded by DARPA CRASH under contract #N66001-10-2-4088.

10. References

10.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <http://www.rfc-editor.org/info/rfc793>.

[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>.

10.2. Informative References

[I-D.briscoe-tcpm-inspace-mode-tcpbis] Briscoe, B., "Inner Space for all TCP Options (Kitchen Sink Draft - to be Split Up)", draft-briscoe-tcpm-inspace-mode-tcpbis-00 (work in progress), March 2015.

[I-D.ietf-tcpm-tcp-edo] Touch, J. and W. Eddy, "TCP Extended Data Offset Option", draft-ietf-tcpm-tcp-edo-03 (work in progress), April 2015.

[I-D.ietf-tls-tls13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-07 (work in progress), July 2015.

[I-D.touch-tcpm-tcp-syn-ext-opt] Touch, J. and T. Faber, "TCP SYN Extended Option Space Using an Out-of-Band Segment", draft-touch-tcpm-tcp-syn-ext-opt-02 (work in progress), April 2015.

[RFC3493] Gilligan, R., Thomson, S., Bound, J., McCann, J., and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC 3493, DOI 10.17487/RFC3493, February 2003, <http://www.rfc-editor.org/info/rfc3493>.

[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>.

[RFC5382] Guha, S., Ed., Biswas, K., Ford, B., Sivakumar, S., and P. Srisuresh, "NAT Behavioral Requirements for TCP", BCP 142, RFC 5382, DOI 10.17487/RFC5382, October 2008, <http://www.rfc-editor.org/info/rfc5382>.

[RFC6394] Barnes, R., "Use Cases and Requirements for DNS-Based Authentication of Named Entities (DANE)", RFC 6394, DOI 10.17487/RFC6394, October 2011, <http://www.rfc-editor.org/info/rfc6394>.

[RFC6544] Rosenberg, J., Keranen, A., Lowekamp, B., and A. Roach, "TCP Candidates with Interactive Connectivity Establishment (ICE)", RFC 6544, DOI 10.17487/RFC6544, March 2012, <http://www.rfc-editor.org/info/rfc6544>.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <http://www.rfc-editor.org/info/rfc7413>.

Authors' Addresses

Andrea Bittau
Stanford University
353 Serra Mall, Room 288
Stanford, CA 94305
US

Email: bittau@cs.stanford.edu

Dan Boneh
Stanford University
353 Serra Mall, Room 475
Stanford, CA 94305
US

Email: dabo@cs.stanford.edu

Daniel B. Giffin
Stanford University
353 Serra Mall, Room 288
Stanford, CA 94305
US

Email: dbg@scs.stanford.edu

Mark Handley
University College London
Gower St.
London WC1E 6BT
UK

Email: M.Handley@cs.ucl.ac.uk

David Mazieres
Stanford University
353 Serra Mall, Room 290
Stanford, CA 94305
US

Email: dm@uun.org

Eric W. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304
US

Email: eric.smith@kestrel.edu

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: December 31, 2018

A. Bittau
Google
D. Giffin
Stanford University
M. Handley
University College London
D. Mazieres
Stanford University
E. Smith
Kestrel Institute

June 29, 2018

TCP-ENO: Encryption Negotiation Option
draft-ietf-tcpinc-tcpeno-19

Abstract

Despite growing adoption of TLS, a significant fraction of TCP traffic on the Internet remains unencrypted. The persistence of unencrypted traffic can be attributed to at least two factors. First, some legacy protocols lack a signaling mechanism (such as a "STARTTLS" command) by which to convey support for encryption, making incremental deployment impossible. Second, legacy applications themselves cannot always be upgraded, requiring a way to implement encryption transparently entirely within the transport layer. The TCP Encryption Negotiation Option (TCP-ENO) addresses both of these problems through a new TCP option-kind providing out-of-band, fully backward-compatible negotiation of encryption.

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 December 31, 2018.

Table of Contents

1. Requirements language

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.

2. Introduction

Many applications and protocols running on top of TCP today do not encrypt traffic. This failure to encrypt lowers the bar for certain attacks, harming both user privacy and system security. Counteracting the problem demands a minimally intrusive, backward-compatible mechanism for incrementally deploying encryption. The TCP Encryption Negotiation Option (TCP-ENO) specified in this document provides such a mechanism.

Introducing TCP options, extending operating system interfaces to support TCP-level encryption, and extending applications to take advantage of TCP-level encryption all require effort. To the greatest extent possible, the effort invested in realizing TCP-level encryption today needs to remain applicable in the future should the need arise to change encryption strategies. To this end, it is useful to consider two questions separately:

1. How to negotiate the use of encryption at the TCP layer, and

2. How to perform encryption at the TCP layer.

This document addresses question 1 with a new TCP option, ENO. TCP-ENO provides a framework in which two endpoints can agree on a TCP encryption protocol (_TEP_) out of multiple possible TEPs. For future compatibility, TEPs can vary widely in terms of wire format, use of TCP option space, and integration with the TCP header and segmentation. However, ENO abstracts these differences to ensure the introduction of new TEPs can be transparent to applications taking advantage of TCP-level encryption.

Question 2 is addressed by one or more companion TEP specification documents. While current TEPs enable TCP-level traffic encryption today, TCP-ENO ensures that the effort invested to deploy today's TEPs will additionally benefit future ones.

2.1. Design goals

TCP-ENO was designed to achieve the following goals:

1. Enable endpoints to negotiate the use of a separately specified TCP encryption protocol (_TEP_) suitable for either opportunistic security [RFC7435] of arbitrary TCP communications or stronger security of applications willing to perform endpoint authentication.

2. Transparently fall back to unencrypted TCP when not supported by both endpoints.

3. Provide out-of-band signaling through which applications can better take advantage of TCP-level encryption (for instance, by improving authentication mechanisms in the presence of TCP-level encryption).

4. Define a standard negotiation transcript that TEPs can use to defend against tampering with TCP-ENO.

5. Make parsimonious use of TCP option space.

6. Define roles for the two ends of a TCP connection, so as to name each end of a connection for encryption or authentication purposes even following a symmetric simultaneous open.

3. Terminology

Throughout this document, we use the following terms, several of which have more detailed normative descriptions in [RFC0793]:

SYN segment
: A TCP segment in which the SYN flag is set

ACK segment
: A TCP segment in which the ACK flag is set (which includes most segments other than an initial SYN segment)

non-SYN segment
: A TCP segment in which the SYN flag is clear

SYN-only segment
: A TCP segment in which the SYN flag is set but the ACK flag is clear

SYN-ACK segment
: A TCP segment in which the SYN and ACK flags are both set

Active opener
: A host that initiates a connection by sending a SYN-only segment. With the BSD socket API, an active opener calls "connect". In client-server configurations, active openers are typically clients.

Passive opener
: A host that does not send a SYN-only segment, but responds to one with a SYN-ACK segment. With the BSD socket API, passive openers call "listen" and "accept", rather than "connect". In client-server configurations, passive openers are typically servers.

Simultaneous open
: The act of symmetrically establishing a TCP connection between two active openers (both of which call "connect" with BSD sockets). Each host of a simultaneous open sends both a SYN-only and a SYN-ACK segment. Simultaneous open is less common than asymmetric open with one active and one passive opener, but can be used for NAT traversal by peer-to-peer applications [RFC5382].

TEP
: A TCP encryption protocol intended for use with TCP-ENO and specified in a separate document.

TEP identifier
: A unique 7-bit value in the range 0x20-0x7f that IANA has assigned to a TEP.

Negotiated TEP
: The single TEP governing a TCP connection, determined by use of the TCP ENO option specified in this document.

4. TCP-ENO Specification

TCP-ENO extends TCP connection establishment to enable encryption opportunistically. It uses a new TCP option-kind [RFC0793] to negotiate one among multiple possible TCP encryption protocols (TEPs). The negotiation involves hosts exchanging sets of supported TEPs, where each TEP is represented by a _suboption_ within a larger TCP ENO option in the offering host’s SYN segment.

If TCP-ENO succeeds, it yields the following information:

- A negotiated TEP, represented by a unique 7-bit TEP identifier,

- A few extra bytes of suboption data from each host, if needed by the TEP,

- o A negotiation transcript with which to mitigate attacks on the negotiation itself,
- o Role assignments designating one endpoint "host A" and the other endpoint "host B", and
- o A bit available to higher-layer protocols at each endpoint for out-of-band negotiation of updated behavior in the presence of TCP encryption.

If TCP-ENO fails, encryption is disabled and the connection falls back to traditional unencrypted TCP.

The remainder of this section provides the normative description of the TCP ENO option and handshake protocol.

4.1. ENO Option

TCP-ENO employs an option in the TCP header [RFC0793]. Figure 1 illustrates the high-level format of this option.

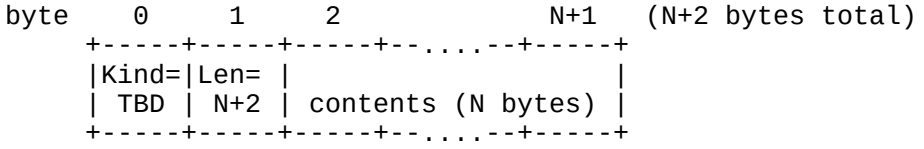


Figure 1: The TCP-ENO option

The contents of an ENO option can take one of two forms. A SYN form, illustrated in Figure 2, appears only in SYN segments. A non-SYN form, illustrated in Figure 3, appears only in non-SYN segments. The SYN form of ENO acts as a container for zero or more suboptions, labeled "Opt_0", "Opt_1", ... in Figure 2. The non-SYN form, by its presence, acts as a one-bit acknowledgment, with the actual contents ignored by ENO. Particular TEPs MAY assign additional meaning to the contents of non-SYN ENO options. When a negotiated TEP does not assign such meaning, the contents of a non-SYN ENO option MUST be zero bytes in sent segments and MUST be ignored in received segments.

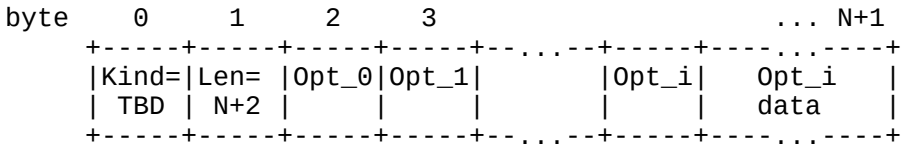


Figure 2: SYN form of ENO

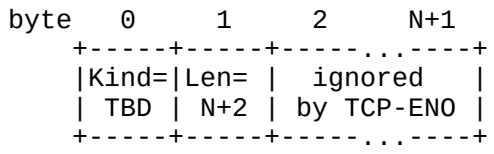


Figure 3: Non-SYN form of ENO, where N MAY be 0

Every suboption starts with a byte of the form illustrated in Figure 4. The high bit "v", when set, introduces suboptions with variable-length data. When "v = 0", the byte itself constitutes the entirety of the suboption. The remaining 7-bit value, called "glt", takes on various meanings, as defined below:

- Global configuration data (discussed in Section 4.2),
- Suboption data length for the next suboption (discussed in Section 4.4), or
- An offer to use a particular TEP defined in a separate TEP specification document.

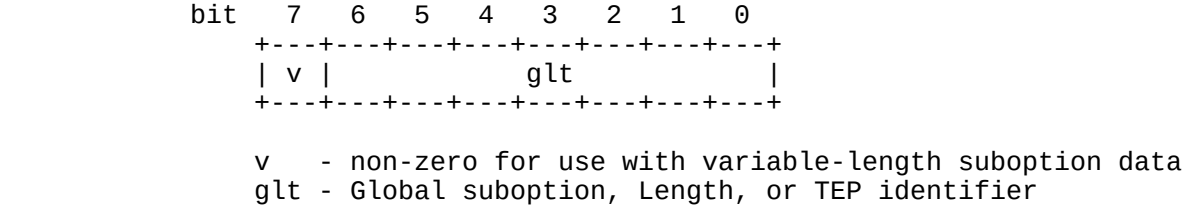


Figure 4: Format of initial suboption byte

Table 1 summarizes the meaning of initial suboption bytes. Values of "glt" below 0x20 are used for global suboptions and length information (the "gl" in "glt"), while those greater than or equal to 0x20 are TEP identifiers (the "t"). When "v = 0", the initial suboption byte constitutes the entirety of the suboption and all information is expressed by the 7-bit "glt" value, which can be either a global suboption or a TEP identifier. When "v = 1", it indicates a suboption with variable-length suboption data. Only TEP identifiers have suboption data, not global suboptions. Hence, bytes with "v = 1" and "glt < 0x20" are not global suboptions but rather length bytes governing the length of the next suboption (which MUST be a TEP identifier). In the absence of a length byte, a TEP identifier suboption with "v = 1" has suboption data extending to the end of the TCP option.

glt	v	Meaning
0x00-0x1f	0	Global suboption (Section 4.2)
0x00-0x1f	1	Length byte (Section 4.4)
0x20-0x7f	0	TEP identifier without suboption data
0x20-0x7f	1	TEP identifier followed by suboption data

Table 1: Initial suboption byte values

A SYN segment MUST contain at most one TCP ENO option. If a SYN segment contains more than one ENO option, the receiver MUST behave as though the segment contained no ENO options and disable encryption. A TEP MAY specify the use of multiple ENO options in a non-SYN segment. For non-SYN segments, ENO itself only distinguishes between the presence or absence of ENO options; multiple ENO options are interpreted the same as one.

4.2. The Global Suboption

Suboptions 0x00-0x1f are used for global configuration that applies regardless of the negotiated TEP. A TCP SYN segment MUST include at most one ENO suboption in this range. A receiver MUST ignore all but the first suboption in this range in any given TCP segment so as to anticipate updates to ENO that assign new meaning to bits in subsequent global suboptions. The value of a global suboption byte is interpreted as a bitmask, illustrated in Figure 5.

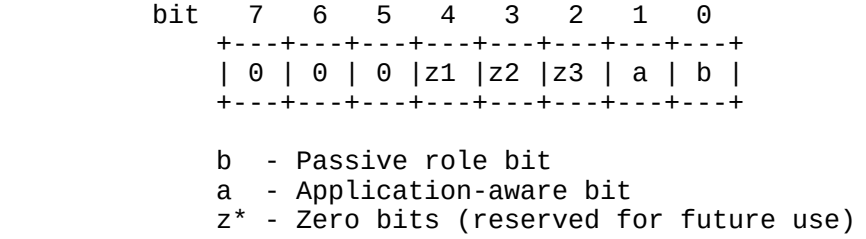


Figure 5: Format of the global suboption byte

The fields of the bitmask are interpreted as follows:

b
: The passive role bit MUST be 1 for all passive openers. For active openers, it MUST default to 0, but implementations MUST provide an API through which an application can explicitly set "b = 1" before initiating an active open. (Manual configuration of "b" is only necessary to enable encryption with a simultaneous

open, and requires prior coordination to ensure exactly one endpoint sets "b = 1" before connecting.)

a
: Legacy applications can benefit from ENO-specific updates that improve endpoint authentication or avoid double encryption. The application-aware bit "a" is an out-of-band signal through which higher-layer protocols can enable ENO-specific updates that would otherwise not be backwards-compatible. Implementations MUST set this bit to 0 by default, and MUST provide an API through which applications can change the value of the bit as well as examine the value of the bit sent by the remote host. Implementations MUST furthermore support a _mandatory_ application-aware mode in which TCP-ENO is automatically disabled if the remote host does not set "a = 1".

z1, z2, z3
: The "z" bits are reserved for future updates to TCP-ENO. They MUST be set to zero in sent segments and MUST be ignored in received segments.

A SYN segment without an explicit global suboption has an implicit global suboption of 0x00. Because passive openers MUST always set "b = 1", they cannot rely on this implicit 0x00 byte and MUST include an explicit global suboption in their SYN-ACK segments.

4.3. TCP-ENO Roles

TCP-ENO uses abstract roles called "A" and "B" to distinguish the two ends of a TCP connection. These roles are determined by the "b" bit in the global suboption. The host that sent an implicit or explicit suboption with "b = 0" plays the A role. The host that sent "b = 1" plays the B role. Because a passive opener MUST set "b = 1" and an active opener by default has "b = 0", the normal case is for the active opener to play role A and the passive opener role B.

Applications performing a simultaneous open, if they desire TCP-level encryption, need to arrange for exactly one endpoint to set "b = 1" (despite being an active opener) while the other endpoint keeps the default "b = 0". Otherwise, if both sides use the default "b = 0" or if both sides set "b = 1", then TCP-ENO will fail and fall back to unencrypted TCP. Likewise, if an active opener explicitly configures "b = 1" and connects to a passive opener (which MUST always have "b = 1"), then TCP-ENO will fail and fall back to unencrypted TCP.

TEP specifications SHOULD refer to TCP-ENO’s A and B roles to specify asymmetric behavior by the two hosts. For the remainder of this

document, we will use the terms "host A" and "host B" to designate the hosts with roles A and B, respectively, in a connection.

4.4. Specifying Suboption Data Length

A TEP MAY optionally make use of one or more bytes of suboption data. The presence of such data is indicated by setting "v = 1" in the initial suboption byte (see Figure 4). A suboption introduced by a TEP identifier with "v = 1" (i.e., a suboption whose first octet has value 0xa0 or higher) extends to the end of the TCP option. Hence, if only one suboption requires data, the most compact way to encode it is to place it last in the ENO option, after all other suboptions. As an example, in Figure 2, the last suboption, "Opt_i", has suboption data and thus requires "v = 1"; however, the suboption data length is inferred from the total length of the TCP option.

When a suboption with data is not last in an ENO option, the sender MUST explicitly specify the suboption data length for the receiver to know where the next suboption starts. The sender does so by introducing the suboption with a length byte, depicted in Figure 6. The length byte encodes a 5-bit value "nnnnn". Adding one to "nnnnn" yields the length of the suboption data (not including the length byte or the TEP identifier). Hence, a length byte can designate anywhere from 1 to 32 bytes of suboption data (inclusive).

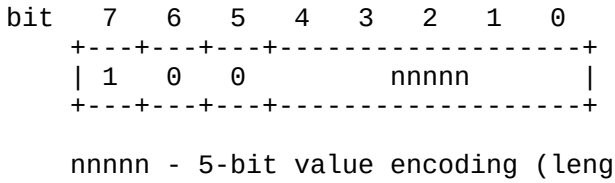


Figure 6: Format of a length byte

A suboption preceded by a length byte MUST be a TEP identifier ("glt >= 0x20") and MUST have "v = 1". Figure 7 shows an example of such a suboption.

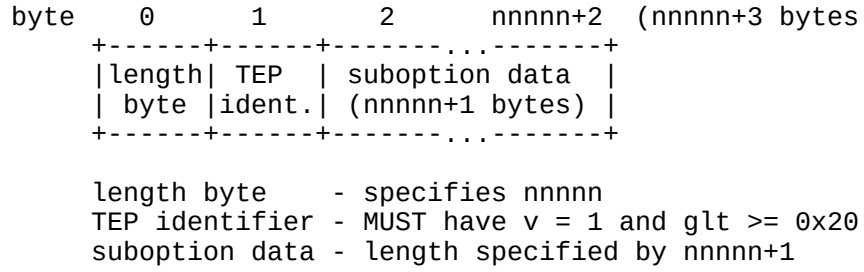


Figure 7: Suboption with length byte

A host MUST ignore an ENO option in a SYN segment and MUST disable encryption if either:

1. A length byte indicates that suboption data would extend beyond the end of the TCP ENO option, or

2. A length byte is followed by an octet in the range 0x00-0x9f (meaning the following byte has "v = 0" or "glt < 0x20").

Because the last suboption in an ENO option is special-cased to have its length inferred from the 8-bit TCP option length, it MAY contain more than 32 bytes of suboption data. Other suboptions are limited to 32 bytes by the length byte format. The TCP header itself can only accommodate a maximum of 40 bytes of options, however. Hence, regardless of the length byte format, a segment would not be able to contain more than one suboption over 32 bytes in size. That said, TEPs MAY define the use of multiple suboptions with the same TEP identifier in the same SYN segment, providing another way to convey over 32 bytes of suboption data even with length bytes.

4.5. The Negotiated TEP

A TEP identifier "glt" (with "glt >= 0x20") is _valid_ for a connection when:

1. Each side has sent a suboption for "glt" in its SYN-form ENO option,

2. Any suboption data in these "glt" suboptions is valid according to the TEP specification and satisfies any runtime constraints, and

3. If an ENO option contains multiple suboptions with "glt", then such repetition is well-defined by the TEP specification.

A passive opener (which is always host B) sees the remote host's SYN segment before constructing its own SYN-ACK segment. Hence, a passive opener SHOULD include only one TEP identifier in SYN-ACK segments and SHOULD ensure this TEP identifier is valid. However, simultaneous open or implementation considerations can prevent host B from offering only one TEP.

To accommodate scenarios in which host B sends multiple TEP identifiers in the SYN-ACK segment, the _negotiated TEP_ is defined as the last valid TEP identifier in host B's SYN-form ENO option. This definition means host B specifies TEP suboptions in order of increasing priority, while host A does not influence TEP priority.

4.6. TCP-ENO Handshake

A host employing TCP-ENO for a connection MUST include an ENO option in every TCP segment sent until either encryption is disabled or the host receives a non-SYN segment. In particular, this means an active opener MUST include a non-SYN-form ENO option in the third segment of a three-way handshake.

A host MUST disable encryption, refrain from sending any further ENO options, and fall back to unencrypted TCP if any of the following occurs:

1. Any segment it receives up to and including the first received ACK segment does not contain a ENO option (or contains an ill-formed SYN-form ENO option),

2. The SYN segment it receives does not contain a valid TEP identifier, or

3. It receives a SYN segment with an incompatible global suboption. (Specifically, incompatible means the two hosts set the same "b" value or the connection is in mandatory application-aware mode and the remote host set "a = 0".)

Hosts MUST NOT alter SYN-form ENO options in retransmitted segments, or between the SYN and SYN-ACK segments of a simultaneous open, with two exceptions for an active opener. First, an active opener MAY unilaterally disable ENO (and thus remove the ENO option) between retransmissions of a SYN-only segment. (Such removal could enable recovery from middleboxes dropping segments with ENO options.) Second, an active opener performing simultaneous open MAY include no TCP-ENO option in its SYN-ACK if the received SYN caused it to disable encryption according to the above rules (for instance because role negotiation failed).

Once a host has both sent and received an ACK segment containing an ENO option, encryption MUST be enabled. Once encryption is enabled, hosts MUST follow the specification of the negotiated TEP and MUST NOT present raw TCP payload data to the application. In particular, data segments MUST NOT contain plaintext application data, but rather ciphertext, key negotiation parameters, or other messages as determined by the negotiated TEP.

A host MAY send a SYN-form ENO option containing zero TEP identifier suboptions, which we term a _vacuous_ ENO option. If either host's SYN segment contains a vacuous ENO option, it follows that there are no valid TEP identifiers for the connection and hence the connection MUST fall back to unencrypted TCP. Hosts MAY send vacuous ENO options to indicate that ENO is supported but unavailable by configuration, or to probe network paths for robustness to ENO options. However, a passive opener MUST NOT send a vacuous ENO option in a SYN-ACK segment unless there was an ENO option in the SYN segment it received. Moreover, a passive opener's SYN-form ENO option MUST still include a global suboption with "b = 1", as discussed in Section 4.3.

4.7. Data in SYN Segments

TEPs MAY specify the use of data in SYN segments so as to reduce the number of round trips required for connection setup. The meaning of data in a SYN segment with an ENO option (a SYN+ENO segment) is determined by the last TEP identifier in the ENO option, which we term the segment's _SYN TEP_. A SYN+ENO segment MAY of course include multiple TEP suboptions, but only the SYN TEP (i.e., the last one) specifies how to interpret the SYN segment's data payload.

A host sending a SYN+ENO segment MUST NOT include data in the segment unless the SYN TEP's specification defines the use of such data. Furthermore, to avoid conflicting interpretations of SYN data, a SYN+ENO segment MUST NOT include a non-empty TCP Fast Open (TFO) option [RFC7413].

Because a host can send SYN data before knowing which if any TEP the connection will negotiate, hosts implementing ENO are REQUIRED to discard data from SYN+ENO segments when the SYN TEP does not become the negotiated TEP. Hosts are furthermore REQUIRED to discard SYN data in cases where another Internet standard specifies a conflicting interpretation of SYN data (as would occur when receiving a non-empty TFO option). This requirement applies to hosts that implement ENO even when ENO has been disabled by configuration. However, note that discarding SYN data is already common practice [RFC4987] and the new requirement applies only to segments containing ENO options.

More specifically, a host that implements ENO MUST discard the data in a received SYN+ENO segment if any of the following applies:

- ENO fails and TEP-indicated encryption is disabled for the connection,
- The received segment's SYN TEP is not the negotiated TEP,
- The negotiated TEP does not define the use of SYN data, or
- The SYN segment contains a non-empty TFO option or any other TCP option implying a conflicting definition of SYN data.

A host discarding SYN data in compliance with the above requirement MUST NOT acknowledge the sequence number of the discarded data, but rather MUST acknowledge the other host's initial sequence number as if the received SYN segment contained no data. Furthermore, after discarding SYN data, such a host MUST NOT assume the SYN data will be identically retransmitted, and MUST process data only from non-SYN segments.

If a host sends a SYN+ENO segment with data and receives acknowledgment for the data, but the SYN TEP in its transmitted SYN segment is not the negotiated TEP (either because a different TEP was negotiated or because ENO failed to negotiate encryption), then the host MUST abort the TCP connection. Proceeding in any other fashion risks misinterpreted SYN data.

If a host sends a SYN-only SYN+ENO segment bearing data and subsequently receives a SYN-ACK segment without an ENO option, that host MUST abort the connection even if the SYN-ACK segment does not acknowledge the SYN data. The issue is that unacknowledged data could nonetheless have been cached by the receiver; later retransmissions intended to supersede this unacknowledged data could fail to do so if the receiver gives precedence to the cached original data. Implementations MAY provide an API call for a non-default mode in which unacknowledged SYN data does not cause a connection abort, but applications MUST use this mode only when a higher-layer integrity check would anyway terminate a garbled connection.

To avoid unexpected connection aborts, ENO implementations MUST disable the use of data in SYN-only segments by default. Such data MAY be enabled by an API command. In particular, implementations MAY provide a per-connection mandatory encryption mode that automatically aborts a connection if ENO fails, and MAY enable SYN data in this mode.

To satisfy the requirement of the previous paragraph, all TEPs SHOULD support a normal mode of operation that avoids data in SYN-only segments. An exception is TEPs intended to be disabled by default.

4.8. Negotiation Transcript

To defend against attacks on encryption negotiation itself, a TEP MUST with high probability fail to establish a working connection between two ENO-compliant hosts when SYN-form ENO options have been altered in transit. (Of course, in the absence of endpoint authentication, two compliant hosts can each still be connected to a man-in-the-middle attacker.) To detect SYN-form ENO option tampering, TEPs MUST reference a transcript of TCP-ENO's negotiation.

TCP-ENO defines its negotiation transcript as a packed data structure consisting of two TCP-ENO options exactly as they appeared in the TCP header (including the TCP option-kind and TCP option length byte as illustrated in Figure 1). The transcript is constructed from the following, in order:

1. The TCP-ENO option in host A's SYN segment, including the kind and length bytes.

2. The TCP-ENO option in host B's SYN segment, including the kind and length bytes.

Note that because the ENO options in the transcript contain length bytes as specified by TCP, the transcript unambiguously delimits A's and B's ENO options.

5. Requirements for TEPs

TCP-ENO affords TEP specifications a large amount of design flexibility. However, to abstract TEP differences away from applications requires fitting them all into a coherent framework. As such, any TEP claiming an ENO TEP identifier MUST satisfy the following normative list of properties.

- TEPs MUST protect TCP data streams with authenticated encryption. (Note "authenticated encryption" refers only to the form of encryption, such as an AEAD algorithm meeting the requirements of [RFC5116]; it does not imply endpoint authentication.)

- TEPs MUST define a session ID whose value identifies the TCP connection and, with overwhelming probability, is unique over all time if either host correctly obeys the TEP. Section 5.1 describes the requirements of the session ID in more detail.

- TEPs MUST NOT make data confidentiality dependent on encryption algorithms with a security strength [SP800-57part1] of less than 120 bits. The number 120 was chosen to accommodate ciphers with 128-bit keys that lose a few bits of security either to particularities of the key schedule or to highly theoretical and unrealistic attacks.

- TEPs MUST NOT allow the negotiation of null cipher suites, even for debugging purposes. (Implementations MAY support debugging modes that allow applications to extract their own session keys.)

- TEPs MUST guarantee the confidentiality of TCP streams without assuming the security of any long-lived secrets. Implementations SHOULD provide forward secrecy soon after the close of a TCP connection, and SHOULD therefore bound the delay between closing a connection and erasing any relevant cryptographic secrets. (Exceptions to forward secrecy are permissible only at the implementation level, and only in response to hardware or architectural constraints--e.g., storage that cannot be securely erased.)

- TEPs MUST protect and authenticate the end-of-file marker conveyed by TCP's FIN flag. In particular, a receiver MUST with overwhelming probability detect a FIN flag that was set or cleared in transit and does not match the sender's intent. A TEP MAY discard a segment with such a corrupted FIN bit, or MAY abort the connection in response to such a segment. However, any such abort MUST raise an error condition distinct from an authentic end-of-file condition.

- TEPs MUST prevent corrupted packets from causing urgent data to be delivered when none has been sent. There are several ways to do so. For instance, a TEP MAY cryptographically protect the URG flag and urgent pointer alongside ordinary payload data. Alternatively, a TEP MAY disable urgent data functionality by clearing the URG flag on all received segments and returning errors in response to sender-side urgent-data API calls. Implementations SHOULD avoid negotiating TEPs that disable urgent data by default. The exception is when applications and protocols are known never to send urgent data.

5.1. Session IDs

Each TEP MUST define a session ID that is computable by both endpoints and uniquely identifies each encrypted TCP connection. Implementations MUST expose the session ID to applications via an API extension. The API extension MUST return an error when no session ID is available because ENO has failed to negotiate encryption or

because no connection is yet established. Applications that are aware of TCP-ENO SHOULD, when practical, authenticate the TCP endpoints by incorporating the values of the session ID and TCP-ENO role (A or B) into higher-layer authentication mechanisms.

In order to avoid replay attacks and prevent authenticated session IDs from being used out of context, session IDs MUST be unique over all time with high probability. This uniqueness property MUST hold even if one end of a connection maliciously manipulates the protocol in an effort to create duplicate session IDs. In other words, it MUST be infeasible for a host, even by violating the TEP specification, to establish two TCP connections with the same session ID to remote hosts properly implementing the TEP.

To prevent session IDs from being confused across TEPs, all session IDs begin with the negotiated TEP identifier--that is, the last valid TEP identifier in host B's SYN segment. Furthermore, this initial byte has bit "v" set to the same value that accompanied the negotiated TEP identifier in B's SYN segment. However, only this single byte is included, not any suboption data. Figure 8 shows the resulting format. This format is designed for TEPs to compute unique identifiers; it is not intended for application authors to pick apart session IDs. Applications SHOULD treat session IDs as monolithic opaque values and SHOULD NOT discard the first byte to shorten identifiers. (An exception is for non-security-relevant purposes, such as gathering statistics about negotiated TEPs.)

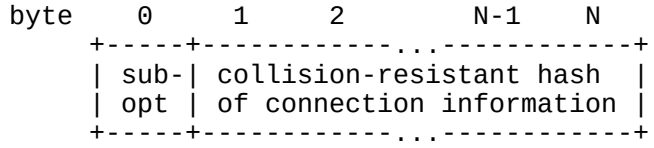


Figure 8: Format of a session ID

Though TEP specifications retain considerable flexibility in their definitions of the session ID, all session IDs MUST meet the following normative list of requirements:

- The session ID MUST be at least 33 bytes (including the one-byte suboption), though TEPs MAY choose longer session IDs.
- The session ID MUST depend in a collision-resistant way on all of the following (meaning it is computationally infeasible to produce collisions of the session ID derivation function unless all of the following quantities are identical):
  - Fresh data contributed by both sides of the connection,

  * Any public keys, public Diffie-Hellman parameters, or other public asymmetric cryptographic parameters that are employed by the TEP and have corresponding private data that is known by only one side of the connection, and

  * The negotiation transcript specified in Section 4.8.

- Unless and until applications disclose information about the session ID, all but the first byte MUST be computationally indistinguishable from random bytes to a network eavesdropper.

- Applications MAY choose to make session IDs public. Therefore, TEPs MUST NOT place any confidential data in the session ID (such as data permitting the derivation of session keys).

6. Examples

This subsection illustrates the TCP-ENO handshake with a few non-normative examples.

```
(1) A -> B:  SYN      ENO<X,Y>
(2) B -> A:  SYN-ACK  ENO<b=1,Y>
(3) A -> B:  ACK      ENO<>
[rest of connection encrypted according to TEP Y]
```

Figure 9: Three-way handshake with successful TCP-ENO negotiation

Figure 9 shows a three-way handshake with a successful TCP-ENO negotiation. Host A includes two ENO suboptions with TEP identifiers X and Y. Host A does not include an explicit global suboption, which means it has an implicit global suboption 0x00 conveying passive role bit "b = 0". The two sides agree to follow the TEP identified by suboption Y.

```
(1) A -> B:  SYN      ENO<X,Y>
(2) B -> A:  SYN-ACK
(3) A -> B:  ACK
[rest of connection unencrypted legacy TCP]
```

Figure 10: Three-way handshake with failed TCP-ENO negotiation

Figure 10 shows a failed TCP-ENO negotiation. The active opener (A) indicates support for TEPs corresponding to suboptions X and Y. Unfortunately, at this point one of several things occurs:

1. The passive opener (B) does not support TCP-ENO,

2. B supports TCP-ENO, but supports neither of TEPs X and Y, and so does not reply with an ENO option,

3. B supports TCP-ENO, but has the connection configured in mandatory application-aware mode and thus disables ENO because A's SYN segment contains an implicit global suboption with "a = 0", or

4. The network stripped the ENO option out of A's SYN segment, so B did not receive it.

Whichever of the above applies, the connection transparently falls back to unencrypted TCP.

```
(1) A -> B:  SYN      ENO<X,Y>
(2) B -> A:  SYN-ACK  ENO<b=1,X> [ENO stripped by middlebox]
(3) A -> B:  ACK
[rest of connection unencrypted legacy TCP]
```

Figure 11: Failed TCP-ENO negotiation because of option stripping

Figure 11 Shows another handshake with a failed encryption negotiation. In this case, the passive opener B receives an ENO option from A and replies. However, the reverse network path from B to A strips ENO options. Hence, A does not receive an ENO option from B, disables ENO, and does not include a non-SYN-form ENO option in segment 3 when ACKing B's SYN. Had A not disabled encryption, Section 4.6 would have required it to include a non-SYN ENO option in segment 3. The omission of this option informs B that encryption negotiation has failed, after which the two hosts proceed with unencrypted TCP.

```
(1) A -> B:  SYN      ENO<Y,X>
(2) B -> A:  SYN      ENO<b=1,X,Y,Z>
(3) A -> B:  SYN-ACK  ENO<Y,X>
(4) B -> A:  SYN-ACK  ENO<b=1,X,Y,Z>
[rest of connection encrypted according to TEP Y]
```

Figure 12: Simultaneous open with successful TCP-ENO negotiation

Figure 12 shows a successful TCP-ENO negotiation with simultaneous open. Here the first four segments contain a SYN-form ENO option, as each side sends both a SYN-only and a SYN-ACK segment. The ENO option in each host's SYN-ACK is identical to the ENO option in its SYN-only segment, as otherwise connection establishment could not recover from the loss of a SYN segment. The last valid TEP in host B's ENO option is Y, so Y is the negotiated TEP.

7. Future Developments

TCP-ENO is designed to capitalize on future developments that could alter trade-offs and change the best approach to TCP-level encryption (beyond introducing new cipher suites). By way of example, we discuss a few such possible developments.

Various proposals exist to increase the maximum space for options in the TCP header. These proposals are highly experimental--particularly those that apply to SYN segments. Hence, future TEPs are unlikely to benefit from extended SYN option space. In the unlikely event that SYN option space is one day extended, however, future TEPs could benefit by embedding key agreement messages directly in SYN segments. Under such usage, the 32-byte limit on length bytes could prove insufficient. This draft intentionally aborts TCP-ENO if a length byte is followed by an octet in the range 0x00-0x9f. If necessary, a future update to this document can define a format for larger suboptions by assigning meaning to such currently undefined byte sequences.

New revisions to socket interfaces [RFC3493] could involve library calls that simultaneously have access to hostname information and an underlying TCP connection. Such an API enables the possibility of authenticating servers transparently to the application, particularly in conjunction with technologies such as DANE [RFC6394]. An update to TCP-ENO can adopt one of the "z" bits in the global suboption to negotiate the use of an endpoint authentication protocol before any application use of the TCP connection. Over time, the consequences of failed or missing endpoint authentication can gradually be increased from issuing log messages to aborting the connection if some as yet unspecified DNS record indicates authentication is mandatory. Through shared library updates, such endpoint authentication can potentially be added transparently to legacy applications without recompilation.

TLS can currently only be added to legacy applications whose protocols accommodate a STARTTLS command or equivalent. TCP-ENO, because it provides out-of-band signaling, opens the possibility of future TLS revisions being generically applicable to any TCP application.

8. Design Rationale

This section describes some of the design rationale behind TCP-ENO.

8.1. Handshake Robustness

Incremental deployment of TCP-ENO depends critically on failure cases devolving to unencrypted TCP rather than causing the entire TCP connection to fail.

Because a network path might drop ENO options in one direction only, a host needs to know not just that the peer supports encryption, but that the peer has received an ENO option. To this end, ENO disables encryption unless it receives an ACK segment bearing an ENO option. To stay robust in the face of dropped segments, hosts continue to include non-SYN form ENO options in segments until such point as they have received a non-SYN segment from the other side.

One particularly pernicious middlebox behavior found in the wild is load balancers that echo unknown TCP options found in SYN segments back to an active opener. The passive role bit "b" in global suboptions ensures encryption will always be disabled under such circumstances, as sending back a verbatim copy of an active opener’s SYN-form ENO option always causes role negotiation to fail.

8.2. Suboption Data

TEPs can employ suboption data for session caching, cipher suite negotiation, or other purposes. However, TCP currently limits total option space consumed by all options to only 40 bytes, making it impractical to have many suboptions with data. For this reason, ENO optimizes the case of a single suboption with data by inferring the length of the last suboption from the TCP option length. Doing so saves one byte.

8.3. Passive Role Bit

TCP-ENO, TEPs, and applications all have asymmetries that require an unambiguous way to identify one of the two connection endpoints. As an example, Section 4.8 specifies that host A’s ENO option comes before host B’s in the negotiation transcript. As another example, an application might need to authenticate one end of a TCP connection with a digital signature. To ensure the signed message cannot not be interpreted out of context to authenticate the other end, the signed message would need to include both the session ID and the local role, A or B.

A normal TCP three-way handshake involves one active and one passive opener. This asymmetry is captured by the default configuration of the "b" bit in the global suboption. With simultaneous open, both hosts are active openers, so TCP-ENO requires that one host explicitly configure "b = 1". An alternate design might

automatically break the symmetry to avoid this need for explicit configuration. However, all such designs we considered either lacked robustness or consumed precious bytes of SYN option space even in the absence of simultaneous open. (One complicating factor is that TCP does not know it is participating in a simultaneous open until after it has sent a SYN segment. Moreover, with packet loss, one host might never learn it has participated in a simultaneous open.)

8.4. Application-aware Bit

Applications developed before TCP-ENO can potentially evolve to take advantage of TCP-level encryption. For instance, an application designed to run only on trusted networks might leverage TCP-ENO to run on untrusted networks, but, importantly, needs to authenticate endpoints and session IDs to do so. In addition to user-visible changes such as requesting credentials, this kind of authentication functionality requires application-layer protocol changes. Some protocols can accommodate the requisite changes--for instance by introducing a new verb analogous to "STARTTLS"--while others cannot do so in a backwards-compatible manner.

The application-aware bit "a" in the the global suboption provides a means of incrementally deploying TCP-ENO-specific enhancements to application-layer protocols that would otherwise lack the necessary extensibility. Software implementing the enhancement always sets "a = 1" in its own global suboption, but only activates the new behavior when the other end of the connection also sets "a = 1".

A related issue is that an application might leverage TCP-ENO as a replacement for legacy application-layer encryption. In this scenario, if both endpoints support TCP-ENO, then application-layer encryption can be disabled in favor of simply authenticating the TCP-ENO session ID. On the other hand, if one endpoint is not aware of the new TCP-ENO-specific mode of operation, there is little benefit to performing redundant encryption at the TCP layer; data is already encrypted once at the application layer, and authentication is only with respect to this application-layer encryption. The mandatory application-aware mode lets applications avoid double encryption in this case: the mode sets "a = 1" in the local host’s global suboption, but also disables TCP-ENO entirely in the event that the other side has not also set "a = 1".

Note that the application-aware bit is not needed by applications that already support adequate higher-layer encryption, such as provided by TLS [RFC5246] or SSH [RFC4253]. To avoid double-encryption in such cases, it suffices to disable TCP-ENO by configuration on any ports with known secure protocols.

8.5. Use of ENO Option Kind by TEPs

This draft does not specify the use of ENO options beyond the first few segments of a connection. Moreover, it does not specify the content of ENO options in non-SYN segments, only their presence. As a result, any use of option-kind TBD after the SYN exchange does not conflict with this document. Because, in addition, ENO guarantees at most one negotiated TEP per connection, TEPs will not conflict with one another or ENO if they use ENO's option-kind for out-of-band signaling in non-SYN segments.

8.6. Unpredictability of Session IDs

Section 5.1 specifies that all but the first (TEP identifier) byte of a session ID MUST be computationally indistinguishable from random bytes to a network eavesdropper. This property is easy to ensure under standard assumptions about cryptographic hash functions. Such unpredictability helps security in a broad range of cases. For example, it makes it possible for applications to use a session ID from one connection to authenticate a session ID from another, thereby tying the two connections together. It furthermore helps ensure that TEPs do not trivially subvert the 33-byte minimum length requirement for session IDs by padding shorter session IDs with zeros.

9. Experiments

This document has experimental status because TCP-ENO's viability depends on middlebox behavior that can only be determined _a posteriori_. Specifically, we need to determine to what extent middleboxes will permit the use of TCP-ENO. Once TCP-ENO is deployed, we will be in a better position to gather data on two types of failure:

1. Middleboxes downgrading TCP-ENO connections to unencrypted TCP. This can happen if middleboxes strip unknown TCP options or if they terminate TCP connections and relay data back and forth.

2. Middleboxes causing TCP-ENO connections to fail completely. This can happen if middleboxes perform deep packet inspection and start dropping segments that unexpectedly contain ciphertext, or if middleboxes strip ENO options from non-SYN segments after allowing them in SYN segments.

Type-1 failures are tolerable, since TCP-ENO is designed for incremental deployment anyway. Type-2 failures are more problematic, and, if prevalent, will require the development of techniques to avoid and recover from such failures. The experiment will succeed so

long as we can avoid type-2 failures and find sufficient use cases that avoid type-1 failures (possibly along with a gradual path for further reducing type-1 failures).

In addition to the question of basic viability, deploying TCP-ENO will allow us to identify and address other potential corner cases or relaxations. For example, does the slight decrease in effective TCP segment payload pose a problem to any applications, requiring restrictions on how TEPs interpret socket buffer sizes? Conversely, can we relax the prohibition on default TEPs that disable urgent data?

A final important metric, related to the pace of deployment and incidence of type-1 failures, will be the extent to which applications adopt TCP-ENO-specific enhancements for endpoint authentication.

10. Security Considerations

An obvious use case for TCP-ENO is opportunistic encryption--that is, encrypting some connections, but only where supported and without any kind of endpoint authentication. Opportunistic encryption provides a property known as _opportunistic security_ [RFC7435], which protects against undetectable large-scale eavesdropping. However, it does not protect against detectable large-scale eavesdropping (for instance, if ISPs terminate TCP connections and proxy them, or simply downgrade connections to unencrypted). Moreover, opportunistic encryption emphatically does not protect against targeted attacks that employ trivial spoofing to redirect a specific high-value connection to a man-in-the-middle attacker. Hence, the mere presence of TEP-indicated encryption does not suffice for an application to represent a connection as "secure" to the user.

Achieving stronger security with TCP-ENO requires verifying session IDs. Any application relying on ENO for communications security MUST incorporate session IDs into its endpoint authentication. By way of example, an authentication mechanism based on keyed digests (such as Digest Access Authentication [RFC7616]) can be extended to include the role and session ID in the input of the keyed digest. Authentication mechanisms with a notion of channel binding (such as SCRAM [RFC5802]) can be updated to derive a channel binding from the session ID. Higher-layer protocols MAY use the application-aware "a" bit to negotiate the inclusion of session IDs in authentication even when there is no in-band way to carry out such a negotiation. Because there is only one "a" bit, however, a protocol extension that specifies use of the "a" bit will likely require a built-in versioning or negotiation mechanism to accommodate crypto agility and future updates.

Because TCP-ENO enables multiple different TEPs to coexist, security could potentially be only as strong as the weakest available TEP. In particular, if TEPs use a weak hash function to incorporate the TCP-ENO transcript into session IDs, then an attacker can undetectably tamper with ENO options to force negotiation of a deprecated and vulnerable TEP. To avoid such problems, security reviewers of new TEPs SHOULD pay particular attention to the collision resistance of hash functions used for session IDs (including the state of cryptanalysis and research into possible attacks). Even if other parts of a TEP rely on more esoteric cryptography that turns out to be vulnerable, it ought nonetheless to be intractable for an attacker to induce identical session IDs at both ends after tampering with ENO contents in SYN segments.

Implementations MUST NOT send ENO options unless they have access to an adequate source of randomness [RFC4086]. Without secret unpredictable data at both ends of a connection, it is impossible for TEPs to achieve confidentiality and forward secrecy. Because systems typically have very little entropy on bootup, implementations might need to disable TCP-ENO until after system initialization.

With a regular three-way handshake (meaning no simultaneous open), the non-SYN form ENO option in an active opener's first ACK segment MAY contain N > 0 bytes of TEP-specific data, as shown in Figure 3. Such data is not part of the TCP-ENO negotiation transcript, and hence MUST be separately authenticated by the TEP.

11. IANA Considerations

[RFC-editor: please replace TBD in this section, in Section 4.1, and in Section 8.5 with the assigned option-kind number. Please also replace RFC-TBD with this document's final RFC number.]

This document defines a new TCP option-kind for TCP-ENO, assigned a value of TBD from the TCP option space. This value is defined as:

Kind	Length	Meaning	Reference
TBD	N	Encryption Negotiation (TCP-ENO)	[RFC-TBD]

TCP Option Kind Numbers

Early implementations of TCP-ENO and a predecessor TCP encryption protocol made unauthorized use of TCP option-kind 69.

[RFC-editor: please glue the following text to the previous paragraph iff TBD == 69, otherwise delete it.] These earlier uses of option 69 are not compatible with TCP-ENO and could disable encryption or suffer complete connection failure when interoperating with TCP-ENO-compliant hosts. Hence, legacy use of option 69 MUST be disabled on hosts that cannot be upgraded to TCP-ENO.

[RFC-editor: please glue this to the previous paragraph regardless of the value of TBD.] More recent implementations used experimental option 253 per [RFC6994] with 16-bit ExID 0x454E. Current and new implementations of TCP-ENO MUST use option TBD, while any legacy implementations MUST migrate to option TBD. Note in particular that Section 4.1 requires at most one SYN-form ENO option per segment, which means hosts MUST NOT not include both option TBD and option 253 with ExID 0x454E in the same TCP segment.

[IANA is also requested to update the entry for TCP-ENO in the TCP Experimental Option Experiment Identifiers (TCP ExIDs) sub-registry to reflect the guidance of the previous paragraph by adding a note saying "current and new implementations MUST use option TDB." RFC-editor: please remove this comment.]

This document defines a 7-bit "glt" field in the range of 0x20-0x7f, for which IANA is to create and maintain a new registry entitled "TCP encryption protocol identifiers" under the "Transmission Control Protocol (TCP) Parameters" registry. The initial contents of the TCP encryption protocol identifier registry is shown in Table 2. This document allocates one TEP identifier (0x20) for experimental use. In case the TEP identifier space proves too small, identifiers in the range 0x70-0x7f are reserved to enable a future update to this document to define extended identifier values. Future assignments are to be made upon satisfying either of two policies defined in [RFC8126]: "IETF Review" or (for non-IETF stream specifications) "Expert Review with RFC Required." IANA will furthermore provide early allocation [RFC7120] to facilitate testing before RFCs are finalized.

Value	Meaning	Reference
0x20	Experimental Use	[RFC-TBD]
0x70-0x7f	Reserved for extended values	[RFC-TBD]

Table 2: TCP encryption protocol identifiers

12. Acknowledgments

We are grateful for contributions, help, discussions, and feedback from the IETF and its TCPINC working group, including Marcelo Bagnulo, David Black, Bob Briscoe, Benoit Claise, Spencer Dawkins, Jake Holland, Jana Iyengar, Tero Kivinen, Mirja Kuhlewind, Watson Ladd, Kathleen Moriarty, Yoav Nir, Christoph Paasch, Eric Rescorla, Adam Roach, Kyle Rose, Michael Scharf, Joe Touch, and Eric Vyncke. This work was partially funded by DARPA CRASH and the Stanford Secure Internet of Things Project.

13. Contributors

Dan Boneh was a co-author of the draft that became this document.

14. References

14.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[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>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[RFC7120] Cotton, M., "Early IANA Allocation of Standards Track Code Points", BCP 100, RFC 7120, DOI 10.17487/RFC7120, January 2014, <https://www.rfc-editor.org/info/rfc7120>.

[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>.

[SP800-57part1]
Barker, E., "Recommendation for Key Management, Part 1: General", NIST Special Publication 800-57 Part 1, Revision 4, January 2016, <http://dx.doi.org/10.6028/NIST.SP.800-57pt1r4>.

14.2. Informative References

[RFC3493] Gilligan, R., Thomson, S., Bound, J., McCann, J., and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC 3493, DOI 10.17487/RFC3493, February 2003, <https://www.rfc-editor.org/info/rfc3493>.

[RFC4253] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, DOI 10.17487/RFC4253, January 2006, <https://www.rfc-editor.org/info/rfc4253>.

[RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, DOI 10.17487/RFC4987, August 2007, <https://www.rfc-editor.org/info/rfc4987>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://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, <https://www.rfc-editor.org/info/rfc5246>.

[RFC5382] Guha, S., Ed., Biswas, K., Ford, B., Sivakumar, S., and P. Srisuresh, "NAT Behavioral Requirements for TCP", BCP 142, RFC 5382, DOI 10.17487/RFC5382, October 2008, <https://www.rfc-editor.org/info/rfc5382>.

[RFC5802] Newman, C., Menon-Sen, A., Melnikov, A., and N. Williams, "Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms", RFC 5802, DOI 10.17487/RFC5802, July 2010, <https://www.rfc-editor.org/info/rfc5802>.

[RFC6394] Barnes, R., "Use Cases and Requirements for DNS-Based Authentication of Named Entities (DANE)", RFC 6394, DOI 10.17487/RFC6394, October 2011, <https://www.rfc-editor.org/info/rfc6394>.

[RFC6994] Touch, J., "Shared Use of Experimental TCP Options", RFC 6994, DOI 10.17487/RFC6994, August 2013, <https://www.rfc-editor.org/info/rfc6994>.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <https://www.rfc-editor.org/info/rfc7413>.

[RFC7435] Dukhovni, V., "Opportunistic Security: Some Protection Most of the Time", RFC 7435, DOI 10.17487/RFC7435, December 2014, <https://www.rfc-editor.org/info/rfc7435>.

[RFC7616] Shekh-Yusef, R., Ed., Ahrens, D., and S. Bremer, "HTTP Digest Access Authentication", RFC 7616, DOI 10.17487/RFC7616, September 2015, <https://www.rfc-editor.org/info/rfc7616>.

Authors' Addresses

Andrea Bittau
Google
345 Spear Street
San Francisco, CA 94105
US

Email: bittau@google.com

Daniel B. Giffin
Stanford University
353 Serra Mall, Room 288
Stanford, CA 94305
US

Email: dbg@scs.stanford.edu

Mark Handley
University College London
Gower St.
London WC1E 6BT
UK

Email: M.Handley@cs.ucl.ac.uk

David Mazieres
Stanford University
353 Serra Mall, Room 290
Stanford, CA 94305
US

Email: dm@uun.org

Eric W. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304
US

Email: eric.smith@kestrel.edu

TCPING
Internet-Draft
Intended status: Standards Track
Expires: May 6, 2016

E. Rescorla
Mozilla

November 3, 2015

Using TLS to Protect TCP Streams
draft-ietf-tcpinc-use-tls-00

Abstract

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

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 May 6, 2016.

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.

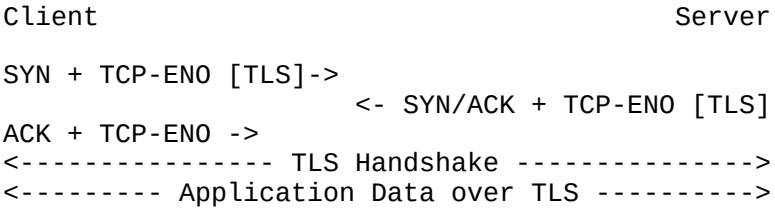


Figure 1

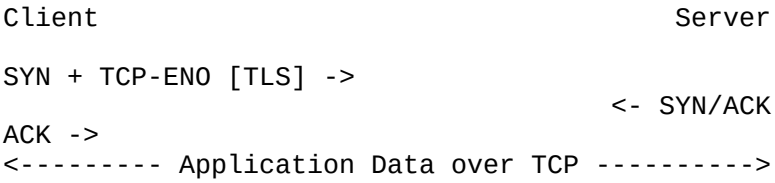


Figure 2: Fall back to TCP

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].

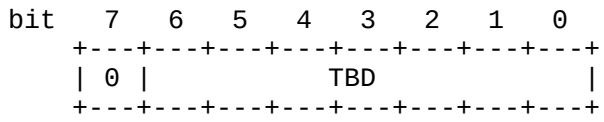
Once the TLS handshake has completed, all application data SHALL be sent over that negotiated TLS channel. Application data MUST NOT be sent prior to the TLS handshake.

If the TLS handshake fails, the endpoint MUST tear down the TCP connection and MUST NOT send plaintext data over the connection.

3. TCP-ENO Binding

3.1. Suboption Definition

TCP-ENO suboption with cs value set to [TBD]. Specifically, this means that the SYN contains a 1-byte suboption indicating support for this specification.



[[OPEN ISSUE: It would be nice to indicate the desire to have 0-RTT, but that would require a variable length suboption, which seems perhaps excessive. Maybe that's the right answer anyway.]]

The SYN/ACK can be in one of two forms:

- A 1-byte suboption as in the SYN.
- A variable-length suboption. In this case, the remainder of the option contains a nonce to be used for 0-RTT (see Section 4.1.4. This nonce MUST be globally unique. Servers MUST NOT use this form of the suboption unless explicitly configured (see Section 6). [[OPEN ISSUE: I just thought this up recently, so it's possible it's totally half-baked and won't work. In particular, am I chewing up too much option space?]]

The ACK simply contains the bare TCP-ENO suboption.

3.2. 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 without a context value (the TCP-ENO transcript is incorporated via the TCPENOTranscript extension).

3.3. Channel Close

Because TLS security is provided in the TCP transport stream rather than at the segment level, the FIN is not an authenticated indicator of end of data. Instead implementations following this specification MUST send a TLS close_notify alert prior to sending a FIN and MUST raise an error if a FIN or RST is receive prior to receiving a close_notify.

4. 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 3.1) but encourage implementations to implement only TLS 1.3. Implementations which also implement TLS 1.2 MUST implement the profile described in Section 4.2

4.1. 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 normative 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.

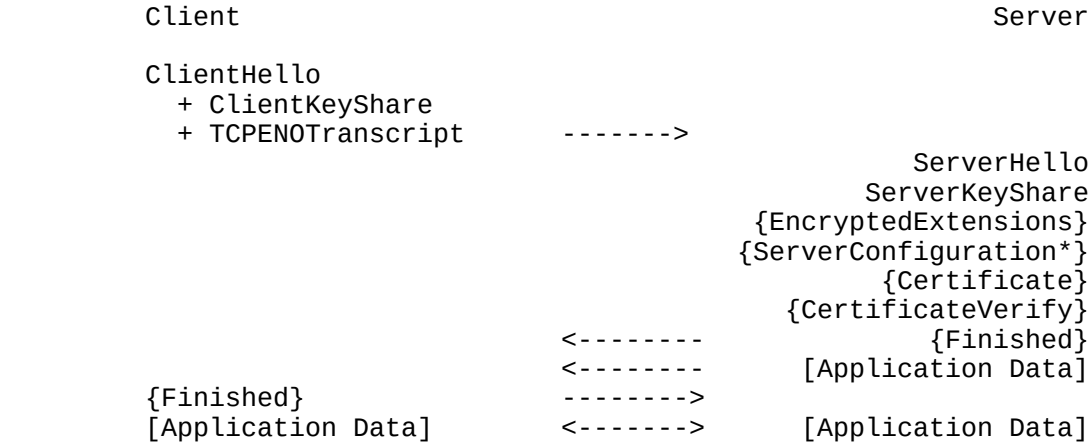
4.1.1. Handshake Modes

TLS 1.3 as used in TCPINC supports two handshake modes, both based on Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) key exchange.

- A 1-RTT mode which is used when the client has no information about the server’s keying material (see Figure 3)
- 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 4)

In both case, the server is expected to have an Elliptic-Curve Digital Signature Algorithm (ECDSA) signing key which may either be a freshly-generated key or a long-term key (allowing Trust-On-First-Use (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.



* 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 3: 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.

4.1.2. Basic 1-RTT Handshake

4.1.2.1. Client’s First Flight

4.1.2.1.1. 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.

The client MUST include a TCPENOTranscript extension containing the TCP-ENO options that were used to negotiate ENO.

4.1.2.2. The TCPENOTranscript

TCPENOTranscript TLS Extension is used to carry the TCP ENO negotiation transcript. The body of the extension simply includes the TCP-ENO negotiation transcript as defined in TCP-ENO Section 3.4.

This serves two purposes:

- It binds the TCP-ENO negotiation into the TLS handshake.
- In 0-RTT mode (see Section 4.1.4) it allows the server to provide an anti-replay nonce which is then mixed into the TLS handshake.

The server MUST validate that the TCPENOTranscript extension matches the transcript. If not, it MUST fail the handshake with a fatal "handshake_failure" exception.

4.1.2.2.1. 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 Section 4.1.2.3). Otherwise, it replies with a HelloRetryRequest (Section 4.1.3), 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.

4.1.2.3. Server's First Flight

4.1.2.3.1. Sending

The server responds 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 4.1.2.1.1 and MUST also ignore any extensions it does not recognize; this implies that the server can implement solely the extensions listed in Section 4.1.2.1.1.

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 4.1.4).

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.

4.1.2.4. Receiving

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

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

- 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.

4.1.2.5. 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.

4.1.3. 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".

4.1.4. 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 4.3), the only valid value here is "early_data", indicating that the client can send data in 0-RTT.

In a future connection, a client MAY send 0-RTT data only if the following three conditions obtain:

- It has been specifically configured to do so (see Section 6).
- A ServerConfiguration is available.
- The server supplied a nonce in its SYN/ACK suboption [[TODO: Work out how to make this work with TFO if at all.]]

In this case, the client sends an EarlyDataIndication extension in its ClientHello and can start sending data immediately, as shown below.

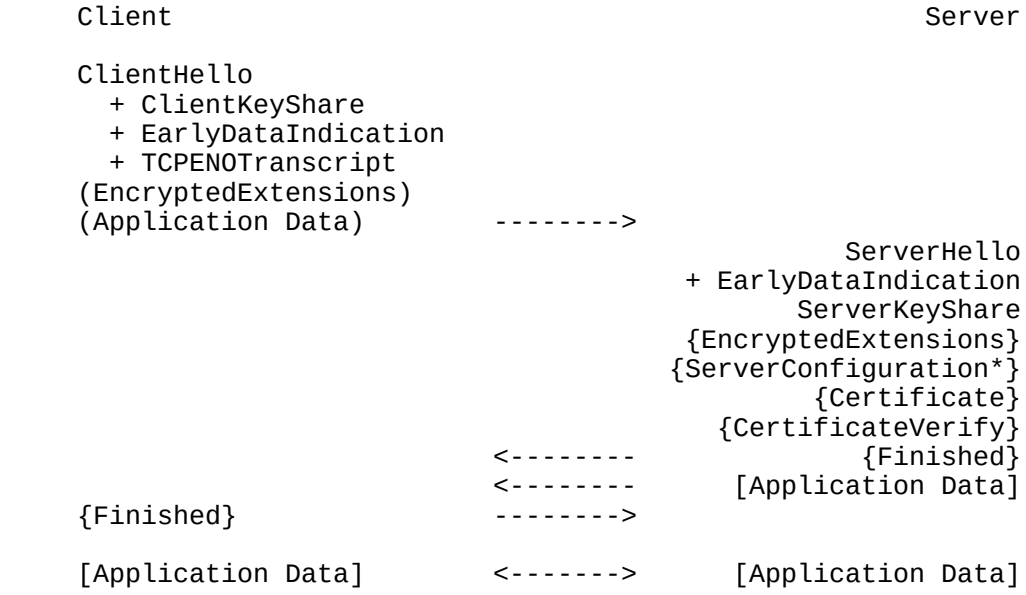


Figure 4: Message flow for a zero round trip handshake

IMPORTANT NOTE: TLS 1.3 Zero-RTT does not provide PFS and therefore MUST only be used when explicitly configured.

Note: TLS 1.3 Zero-RTT data is inherently replayable (see the note in [I-D.ietf-tls-tls13] Section 6.2.2). However, because the client and server have already exchanged data in the _TCP_ handshake, this data can be used to provide anti-replay for a 0-RTT mode TLS handshake via the TCPENOTranscript extension.

4.1.5. 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 the HKDF key-derivation algorithm [RFC5869].

4.1.6. 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 Authenticated Encryption with Additional Data (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 (N_MIN is the minimum nonce size defined in [RFC5116] Section 4) and XORing it with the sequence number of the TLS record (left-padded with zeroes).

- 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 byte value is always non-zero, the padding is removed by removing bytes from the right until a non-zero byte is encountered.

4.2. 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.

4.3. 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

These features have only minimal advantage in this context and interfere with offering a reduced profile.

4.4. 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

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 Application-Layer Protocol Negotiation (ALPN)

[RFC7301] to negotiate the use of AO. [[OPEN ISSUE: Is this something we want? Maybe in a separate specification.]]

6. API Considerations

Needed here:

- How to configure 0-RTT and send 0-RTT data (some sort of sockopt).
- When is the session-id available (post-connect() completion).
- How to indicate that the certificate should be validated.

7. Implementation Considerations

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.

8. 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.

9. 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.

10. 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.

11. References

11.1. 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, "ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)", draft-ietf-tls-chacha20-poly1305-01 (work in progress), November 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-tls13-10 (work in progress), October 2015.

[I-D.irtf-cfrg-curves]
Langley, A. and M. Hamburg, "Elliptic Curves for Security", draft-irtf-cfrg-curves-11 (work in progress), October 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>.

11.2. 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-01 (work in progress), November 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>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <http://www.rfc-editor.org/info/rfc7301>.

Author’s Address

Eric Rescorla
Mozilla

EMail: ekr@rtfm.com

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

E. Rescorla
Mozilla

May 03, 2016

Using TLS to Protect TCP Streams
draft-ietf-tcpinc-use-tls-01

Abstract

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

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 November 4, 2016.

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.

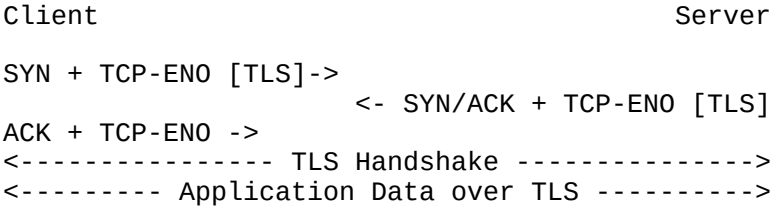


Figure 1

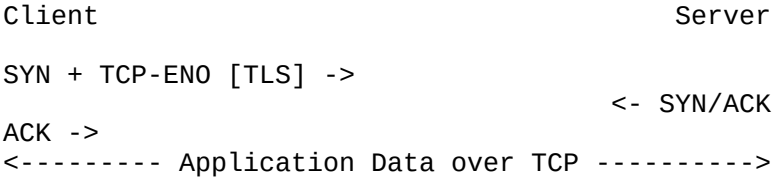


Figure 2: Fall back to TCP

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].

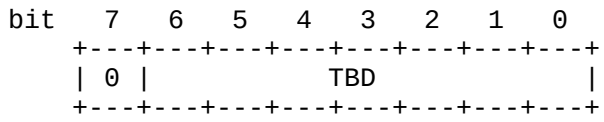
Once the TLS handshake has completed, all application data SHALL be sent over that negotiated TLS channel. Application data MUST NOT be sent prior to the TLS handshake.

If the TLS handshake fails, the endpoint MUST tear down the TCP connection and MUST NOT send plaintext data over the connection.

3. TCP-ENO Binding

3.1. Suboption Definition

TCP-ENO suboption with cs value set to [TBD]. Specifically, this means that the SYN contains a 1-byte suboption indicating support for this specification.



[[OPEN ISSUE: It would be nice to indicate the desire to have 0-RTT, but that would require a variable length suboption, which seems perhaps excessive. Maybe that's the right answer anyway.]]

The SYN/ACK can be in one of two forms:

- A 1-byte suboption as in the SYN.
- A variable-length suboption. In this case, the remainder of the option contains a nonce to be used for 0-RTT (see Section 4.1.4. This nonce MUST be globally unique. Servers MUST NOT use this form of the suboption unless explicitly configured (see Section 6). [[OPEN ISSUE: I just thought this up recently, so it's possible it's totally half-baked and won't work. In particular, am I chewing up too much option space?]]

The ACK simply contains the bare TCP-ENO suboption.

3.2. 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 without a context value (the TCP-ENO transcript is incorporated via the TCPENOTranscript extension).

3.3. Channel Close

Because TLS security is provided in the TCP transport stream rather than at the segment level, the FIN is not an authenticated indicator of end of data. Instead implementations following this specification MUST send a TLS close_notify alert prior to sending a FIN and MUST raise an error if a FIN or RST is receive prior to receiving a close_notify.

4. 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 3.1) but encourage implementations to implement only TLS 1.3. Implementations which also implement TLS 1.2 MUST implement the profile described in Section 4.2

4.1. 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 normative 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.

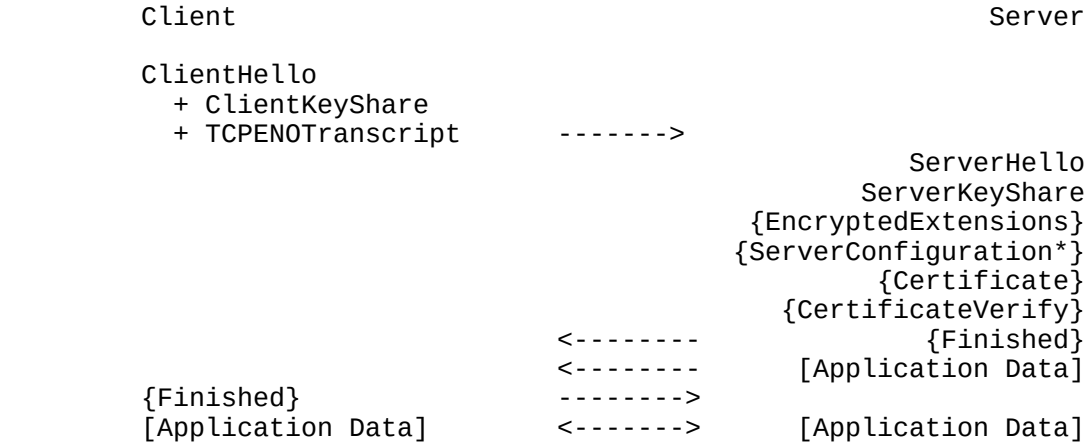
4.1.1. Handshake Modes

TLS 1.3 as used in TCPINC supports two handshake modes, both based on Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) key exchange.

- A 1-RTT mode which is used when the client has no information about the server’s keying material (see Figure 3)
- 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 4)

In both case, the server is expected to have an Elliptic-Curve Digital Signature Algorithm (ECDSA) signing key which may either be a freshly-generated key or a long-term key (allowing Trust-On-First-Use (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.



* 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 3: 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.

4.1.2. Basic 1-RTT Handshake

4.1.2.1. Client’s First Flight

4.1.2.1.1. 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.

The client MUST include a TCPENOTranscript extension containing the TCP-ENO options that were used to negotiate ENO.

4.1.2.2. The TCPENOTranscript

TCPENOTranscript TLS Extension is used to carry the TCP ENO negotiation transcript. The body of the extension simply includes the TCP-ENO negotiation transcript as defined in TCP-ENO Section 3.4.

This serves two purposes:

- It binds the TCP-ENO negotiation into the TLS handshake.
- In 0-RTT mode (see Section 4.1.4) it allows the server to provide an anti-replay nonce which is then mixed into the TLS handshake.

The server MUST validate that the TCPENOTranscript extension matches the transcript. If not, it MUST fail the handshake with a fatal "handshake_failure" exception.

4.1.2.2.1. 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 Section 4.1.2.3). Otherwise, it replies with a HelloRetryRequest (Section 4.1.3), 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.

4.1.2.3. Server's First Flight

4.1.2.3.1. Sending

The server responds 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 4.1.2.1.1 and MUST also ignore any extensions it does not recognize; this implies that the server can implement solely the extensions listed in Section 4.1.2.1.1.

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 4.1.4).

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.

4.1.2.4. Receiving

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

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

- 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.

4.1.2.5. 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.

4.1.3. 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".

4.1.4. 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 4.3), the only valid value here is "early_data", indicating that the client can send data in 0-RTT.

In a future connection, a client MAY send 0-RTT data only if the following three conditions obtain:

- It has been specifically configured to do so (see Section 6).
- A ServerConfiguration is available.
- The server supplied a nonce in its SYN/ACK suboption [[TODO: Work out how to make this work with TFO if at all.]]

In this case, the client sends an EarlyDataIndication extension in its ClientHello and can start sending data immediately, as shown below.

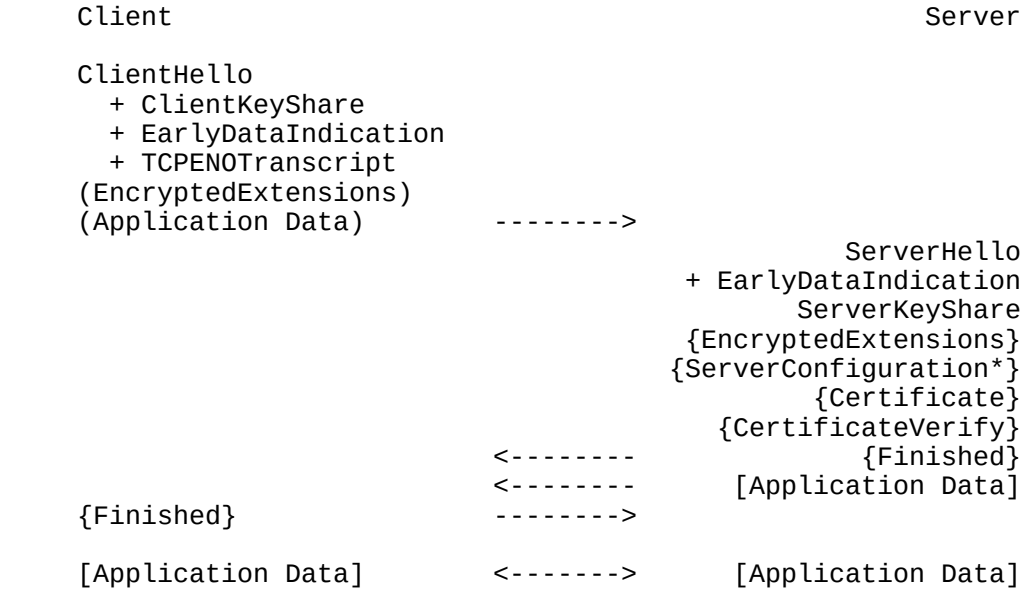


Figure 4: Message flow for a zero round trip handshake

IMPORTANT NOTE: TLS 1.3 Zero-RTT does not provide PFS and therefore MUST only be used when explicitly configured.

Note: TLS 1.3 Zero-RTT data is inherently replayable (see the note in [I-D.ietf-tls-tls13] Section 6.2.2). However, because the client and server have already exchanged data in the _TCP_ handshake, this data can be used to provide anti-replay for a 0-RTT mode TLS handshake via the TCPENOTranscript extension.

4.1.5. 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 the HKDF key-derivation algorithm [RFC5869].

4.1.6. 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 Authenticated Encryption with Additional Data (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 (N_MIN is the minimum nonce size defined in [RFC5116] Section 4) and XORing it with the sequence number of the TLS record (left-padded with zeroes).

- 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 byte value is always non-zero, the padding is removed by removing bytes from the right until a non-zero byte is encountered.

4.2. 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.

4.3. 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

These features have only minimal advantage in this context and interfere with offering a reduced profile.

4.4. 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

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 Application-Layer Protocol Negotiation (ALPN)

[RFC7301] to negotiate the use of AO. [[OPEN ISSUE: Is this something we want? Maybe in a separate specification.]]

6. API Considerations

Needed here:

- How to configure 0-RTT and send 0-RTT data (some sort of sockopt).
- When is the session-id available (post-connect() completion).
- How to indicate that the certificate should be validated.

7. Implementation Considerations

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.

8. 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.

9. 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.

10. 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.

11. References

11.1. 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, "ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)", draft-ietf-tls-chacha20-poly1305-04 (work in progress), December 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-tls13-12 (work in progress), March 2016.

[I-D.irtf-cfrg-curves]
Langley, A. and M. Hamburg, "Elliptic Curves for Security", draft-irtf-cfrg-curves-11 (work in progress), October 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>.

11.2. 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-01 (work in progress), November 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>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <http://www.rfc-editor.org/info/rfc7301>.

Author’s Address

Eric Rescorla
Mozilla

EMail: ekr@rtfm.com