Internet Engineering Task Force (IETF)
Request for Comments: 8548
Category: Experimental
ISSN: 2070-1721

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

May 2019

Cryptographic Protection of TCP Streams (tcpcrypt)

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 document is not an Internet Standards Track specification; it is published for examination, experimental implementation, and evaluation.

This document defines an Experimental Protocol for the Internet community. This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are candidates for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc8548.

Copyright Notice

Table of Contents

1. 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) [RFC8547] 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 [TCPINC-API] describes recommended interfaces for configuring certain parameters of this protocol.

2. 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.

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 the HMAC-based Key Derivation Function (HKDF) [RFC5869], which is built on Keyed-Hashing for Message Authentication (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 [RFC8547] to negotiate whether encryption will be enabled for a connection as well as which key-agreement scheme to use. TCP-ENO negotiates the use of a particular TCP encryption protocol (TEP) by including protocol identifiers in ENO suboptions. This document associates four TEP identifiers with the tcpcrypt protocol as listed in Table 4 of 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 that 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; 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 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, in which case 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 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 of 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 [RFC8547], 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 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 represented by 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 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 of Section 6. The algorithm uses the first ae_key_len bytes of each traffic key as an authenticated-encryption key, and it uses 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 advertise 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 [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 Section 3.8 of [RFC793]. 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 its willingness to resume with a particular session secret by sending a SYN segment with a resumption suboption, i.e., 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:

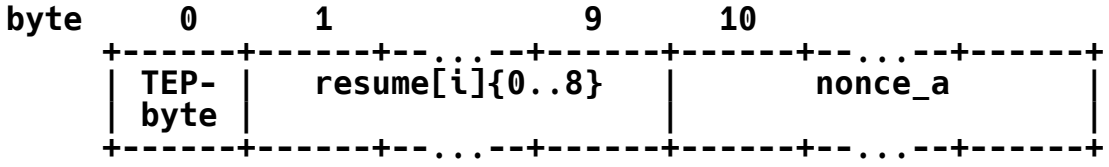


Figure 2: Resumption suboption sent when original role was "A".

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:

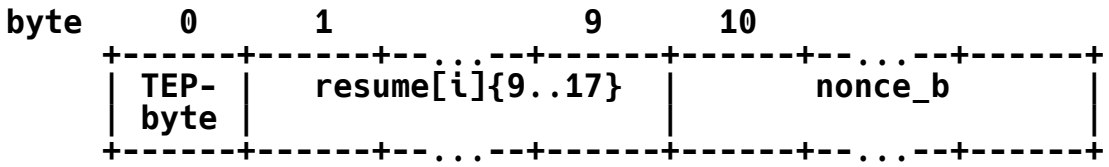


Figure 3: Resumption suboption sent when original role was "B".

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, it 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; 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 as well as from 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 it 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 and 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 with both segments 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 [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 of 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 of Section 6 along 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 field 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 rekeying as described in Section 3.8.

Keys for AEAD encryption are 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 appropriate traffic key is divided into two parts:

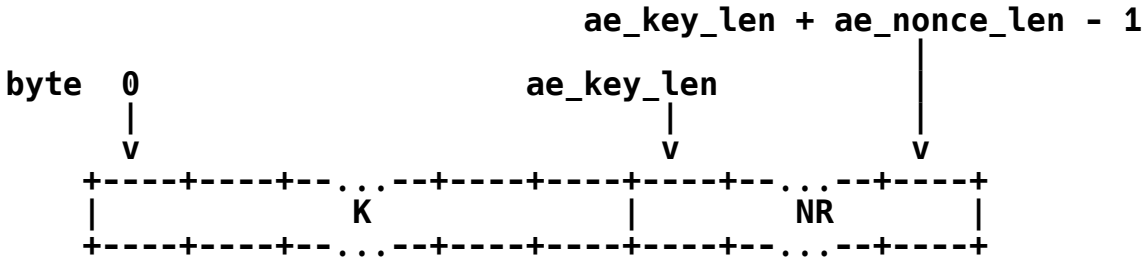


Figure 4: Format of Traffic Key

With reference to the "AEAD Interface" described in Section 2 of [RFC5116], the first ae_key_len bytes of the traffic key provide the AEAD key K. The remaining ae_nonce_len bytes provide a nonce randomizer value NR, which is combined via bitwise exclusive-or with the frame ID to yield N, the AEAD nonce for the frame:

N = frame_ID XOR NR

The remaining AEAD inputs, P and A, are provided by the frame's plaintext value and associated data, respectively. The output of the AEAD 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 selectively acknowledged (SACKed), according to Section 8 of [RFC2018]).

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 field 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 Section 8 of [RFC2018]).

3.8. Rekeying

Rekeying 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 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 both 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 rekeying".

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 rekey operation if it has no data to send; that is, it MUST NOT initiate rekeying 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 rekeying mechanism for this purpose, as follows. When necessary, a host SHOULD probe the liveness of its peer by initiating rekeying 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).

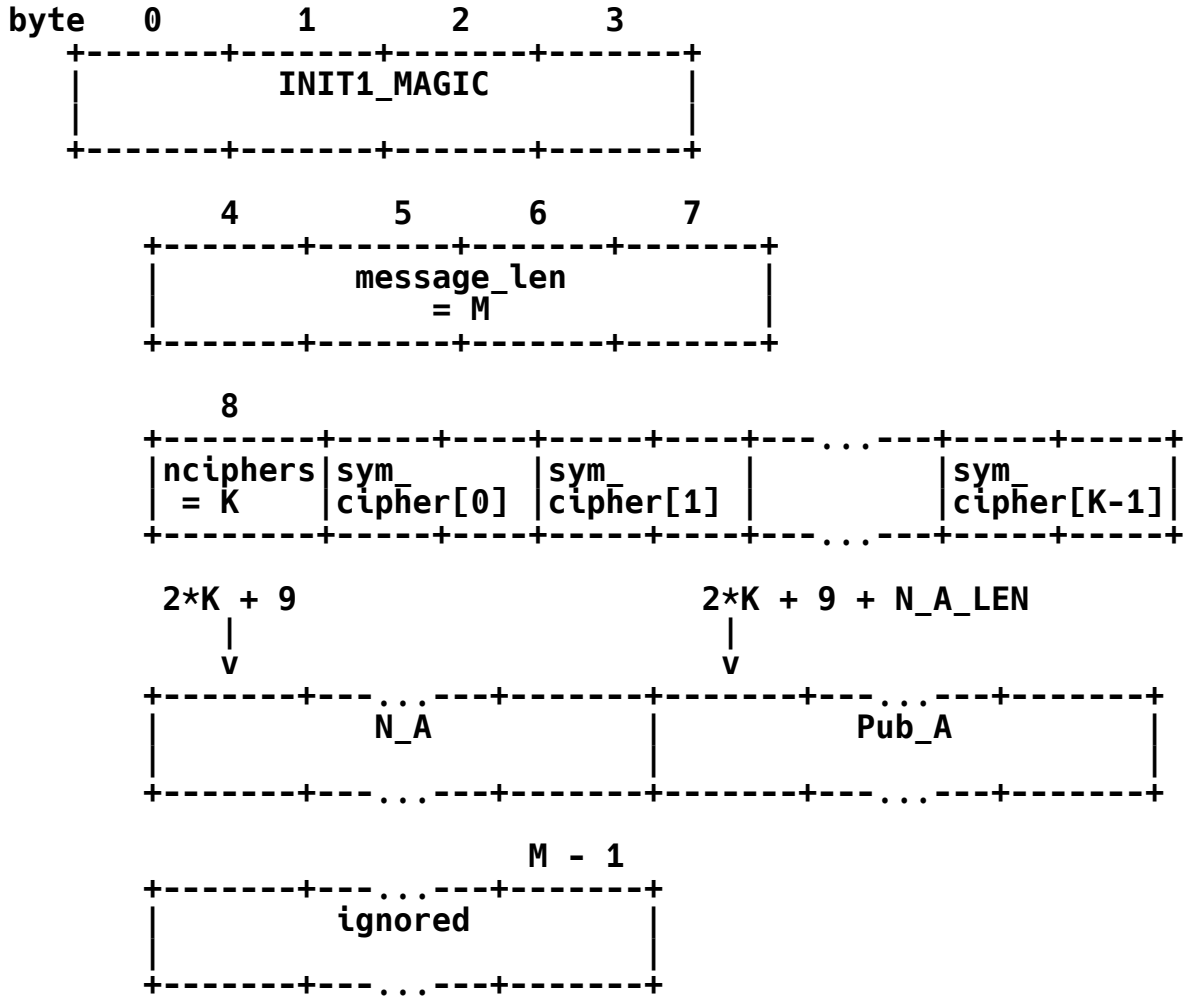
Implementations MAY use TCP keep-alives for purposes that do not require endpoint authentication, as discussed in Section 8.2.

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:

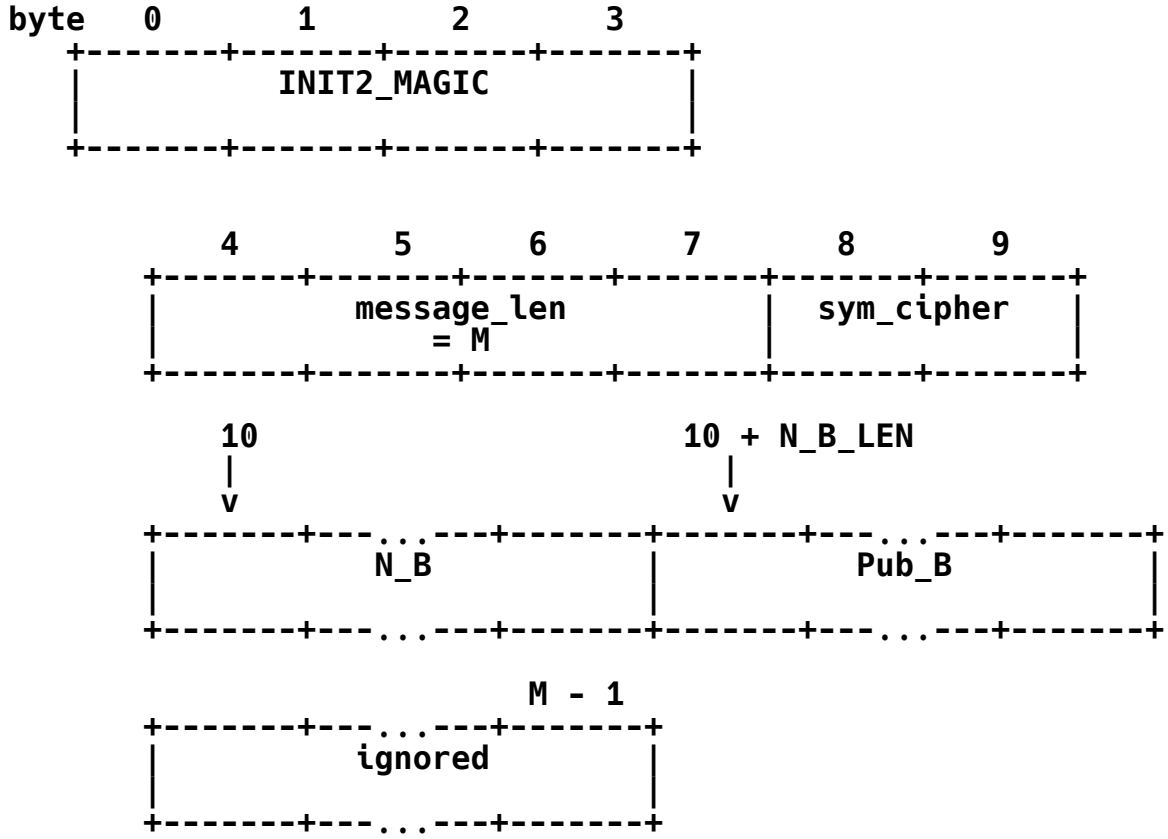


The constant INIT1_MAGIC is defined in Section 4.3. The four-byte field message_len gives the length of the entire Init1 message, encoded as a big-endian integer. The nciphers field contains an integer value that specifies the number of two-byte symmetric-cipher identifiers that follow. The sym_cipher[i] 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 ignored field 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:

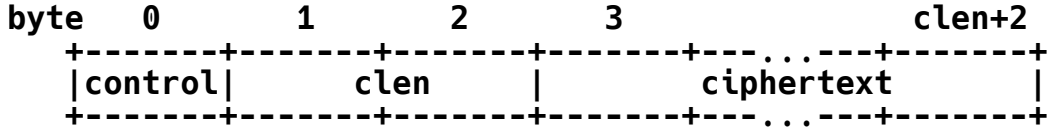


The constant INIT2_MAGIC is defined in Section 4.3. The four-byte field message_len gives the length of the entire Init2 message, encoded as a big-endian integer. The sym_cipher value is a selection from the symmetric-cipher identifiers in the previously-received Init1 message. The length N_B_LEN and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

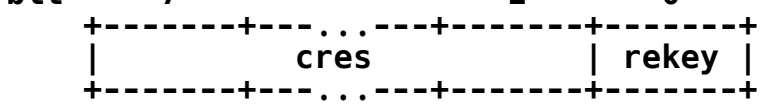
4.2. Encryption Frames

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 control field has this structure:

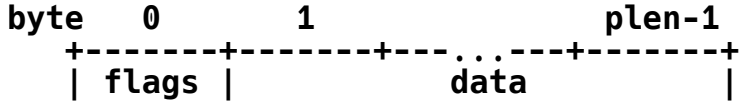


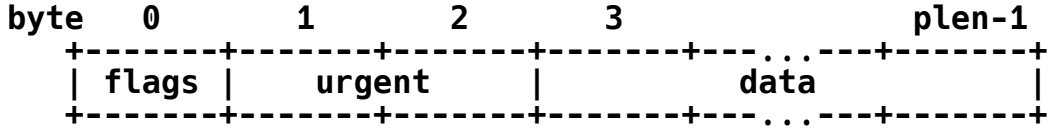
The seven-bit field cres is reserved; implementations MUST set these bits to zero when sending and MUST ignore them when receiving.

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

4.2.1. Plaintext

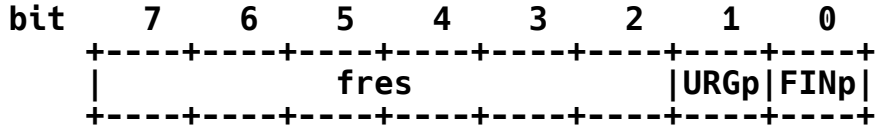
The ciphertext field is the result of applying the negotiated authenticated-encryption algorithm to a plaintext value, which has one of these two formats:





(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 field has this structure:



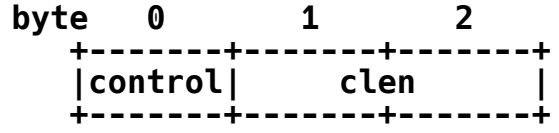
The six-bit field fres is reserved; implementations MUST set these six bits to zero when sending, and MUST ignore them when receiving.

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 the urgent field and of the flag bits is described in Section 3.7.

4.2.2. Associated Data

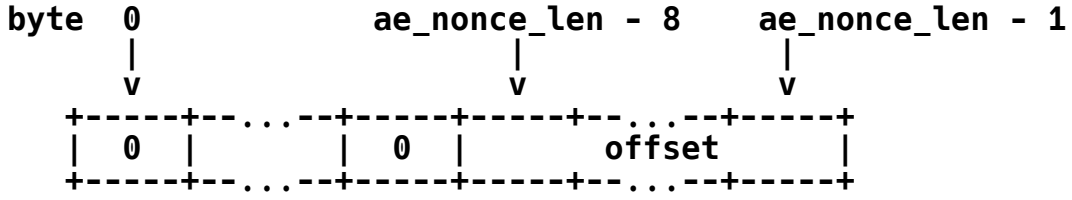
An encryption frame's associated data (which is supplied to the AEAD algorithm when decrypting the ciphertext and verifying the frame's integrity) has this format:



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.

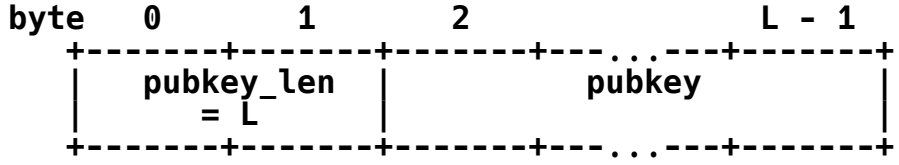
All the TEPs specified in this document require the use of HKDF-Expand-SHA256 as the CPRF, and these lengths for nonces and session secrets:

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

Future documents assigning additional TEPs for use with tcpcrypt might 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 Elliptic Curve Secret Value Derivation Primitive, Diffie-Hellman version

(ECSVDP-DH) 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 Section A.16.10 of [IEEE-1363].

Key-agreement schemes ECDHE-Curve25519 and ECDHE-Curve448 perform the Diffie-Hellman 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 [RFC8439].

Implementations MUST support certain AEAD algorithms according to Table 3. Note that system administrators MAY configure which algorithms a host will negotiate independently 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 have been incorporated in IANA's "TCP Encryption Protocol Identifiers" registry under the "Transmission Control Protocol (TCP) Parameters" registry, as in Table 4. The various key-agreement schemes used by these tcpcrypt variants are defined in Section 5.

Value	Meaning	Reference
0x21	TCPCRYPT_ECDHE_P256	[RFC8548]
0x22	TCPCRYPT_ECDHE_P521	[RFC8548]
0x23	TCPCRYPT_ECDHE_Curve25519	[RFC8548]
0x24	TCPCRYPT_ECDHE_Curve448	[RFC8548]

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, inclusively, for which IANA maintains a new "tcpcrypt AEAD Algorithms" registry under the "Transmission Control Protocol (TCP) Parameters" registry. The initial values for this registry are given in Table 5. 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	[RFC8548], Section 6
0x0002	AEAD_AES_256_GCM	[RFC8548], Section 6
0x0010	AEAD_CHACHA20_POLY1305	[RFC8548], 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., Elliptic Curve Diffie-Hellman (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. Therefore, 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 rekey 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; 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 key agreement based on Ephemeral Elliptic Curve Diffie-Hellman (ECDHE), 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 is 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 it 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. Therefore, 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 Section 9 of [RFC8547], 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 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. References

10.1. Normative References

[IEEE-1363] IEEE, "IEEE Standard Specifications for Public-Key Cryptography", IEEE Standard 1363-2000, DOI 10.1109/IEEESTD.2000.92292.

[NIST-DSS] National Institute of Standards and Technology (NIST), "Digital Signature Standard (DSS)", FIPS PUB 186-4, DOI 10.6028/NIST.FIPS.186-4, July 2013.

[RFC793] 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>.

[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>.

[RFC8439] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <https://www.rfc-editor.org/info/rfc8439>.

[RFC8547] Bittau, A., Giffin, D., Handley, M., Mazieres, D., and E. Smith, "TCP-ENO: Encryption Negotiation Option", RFC 8547, DOI 10.17487/RFC8547, May 2019, <https://www.rfc-editor.org/info/rfc8547>.

10.2. Informative References

[NIST-fail] Bernstein, D. and T. Lange, "Failures in NIST's ECC Standards", January 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 Symposium, August 2010.

[TCPINC-API] Bittau, A., Boneh, D., Giffin, D., Handley, M., Mazieres, D., and E. Smith, "Interface Extensions for TCP-ENO and tcpcrypt", Work in Progress, draft-ietf-tcpinc-api-06, June 2018.

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.

Contributors

Dan Boneh and Michael Hamburg were coauthors of the draft that became this document.

Authors' Addresses

Andrea Bittau
Google
345 Spear Street
San Francisco, CA 94105
United States of America

Email: bittau@google.com

Daniel B. Giffin
Stanford University
353 Serra Mall, Room 288
Stanford, CA 94305
United States of America

Email: daniel@beech-grove.net

Mark Handley
University College London
Gower St.
London WC1E 6BT
United Kingdom

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

David Mazieres
Stanford University
353 Serra Mall, Room 290
Stanford, CA 94305
United States of America

Email: dm@uun.org

Quinn Slack
Sourcegraph
121 2nd St Ste 200
San Francisco, CA 94105
United States of America

Email: sqs@sourcegraph.com

Eric W. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304
United States of America

Email: eric.smith@kestrel.edu