Network Working Group
Internet-Draft
Intended status: Experimental
Expires: May 6, 2018

A. Bittau
Google
D. Giffin
Stanford University
M. Handley
University College London
D. Mazieres
Stanford University
Q. Slack
Sourcegraph
E. Smith
Kestrel Institute
November 2, 2017

# Cryptographic protection of TCP Streams (tcpcrypt)
draft-ietf-tcpinc-tcpcrypt-09

## 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 may be transmitted. However, this cost 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 May 6, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

## 1. 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 tcpcrypt protocol at an abstract level. The concrete format of all messages is specified in Section 4.

### 3.1. Cryptographic Algorithms

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





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

- o An _extract function_ is used to generate a pseudo-random key (PRK) 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.

- o 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. Each identifier indicates the use of a particular key-agreement scheme, with an associated CPRF and length parameters. Future standards may 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", and an identifier derived from a session secret previously negotiated with the same host and the same TEP.

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 it had "v = 1", the key-exchange messages will be omitted in favor of determining keys via session-resumption as described in Section 3.5, and protected application data may immediately be sent 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 may 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, PK_A }
B -> A:  Init2 = { INIT2_MAGIC, sym_cipher, N_B, PK_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 Table 1.





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

- "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.

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

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, "PK_B".

The two sides then compute a pseudo-random key ("PRK"), from which all session keys 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 then 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" 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 1. 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, 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.

Finally, each master key "mk[j]" is used to generate keys for authenticated encryption:

```
k_ab[j] = CPRF(mk[j], CONST_KEY_A, ae_keylen)
k_ba[j] = CPRF(mk[j], CONST_KEY_B, ae_keylen)
```

In the first session derived from fresh key-agreement, 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 value "ae_keylen" depends on the authenticated-encryption algorithm selected, and is given under "Key Length" in Table 5.

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.

As required, 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, as follows:

```
session_id[i] = TEP-byte | CPRF(ss[i], CONST_SESSID, 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 a particular session secret, they can establish a new connection without public-key operations using 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 whose value is the negotiated TEP identifier of the session concatenated with half of the "resumption identifier" for the session.

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:

```
byte     0        1                  9      (10 bytes total)
     +--------+--------+---...---+--------+
     |  TEP-  |      resume[i]{0..8}       |
     |  byte  |                            |
     +--------+--------+---...---+--------+
```

Figure 2: Resumption suboption sent when original role was A. The TEP-byte contains a tcpcrypt TEP identifier and v = 1.

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

```
     byte     0        1                  9      (10 bytes total)
            +--------+--------+---...---+--------+
            |  TEP-  |      resume[i]{9..17}     |
            |  byte  |                           |
            +--------+--------+---...---+--------+
```

Figure 3: Resumption suboption sent when original role was B. The TEP-byte contains a tcpcrypt TEP identifier and v = 1.

If a passive opener receives a resumption suboption containing an identifier-half it recognizes as being derived from a session secret that it has cached, 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.

If the 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 suboption.

If an active opener receives a resumption suboption for a particular TEP and the received identifier-half does not match the "resume[i]" value whose other half it previously sent in a resumption suboption for the same TEP, 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, 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.

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, 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 "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 retain "ss[i+1]" 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.

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 pre-session key "ss[0]".

A session secret may 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 will have to revert to fresh key-exchange. To avoid this limitation, implementations MAY choose to implement session resumption such that a given pre-session key "ss[0]" is only 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 pre-session key "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 be able to specify that during the connection no session secrets will be cached and all resumption requests will be ignored in favor of fresh key exchange. And for an established connection, an application must be able 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 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 5, and additional algorithms may be specified according to the policy in Section 7. One algorithm is selected during the negotiation described in Section 3.3.

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 nonce" value (see Section 4.2.3) is also constructed for the frame but not explicitly transmitted. It contains an "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.) 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 nonce 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.

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[j]" or "k_ba[j]" for some "j", according to the host's role as described in Section 3.3. 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 a plaintext value "P" or the special symbol FAIL. In the latter case, the implementation MUST either drop the TCP segment(s) containing the frame or abort the connection; but if it aborts, the implementation MUST raise an error condition distinct from the end-of-file condition.

### 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.

## 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 this field to zero 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 SHOULD NOT initiate more than one concurrent re-key operation if it has no data to send; that is, it should 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:

```
    byte   0       1       2       3
        +-------+-------+-------+-------+
        |          INIT1_MAGIC          |
        |                               |
        +-------+-------+-------+-------+

            4         5      6       7
            +-------+-------+-------+-------+
            |          message_len          |
            |              = M              |
            +-------+-------+-------+-------+

            8
            +--------+-----+----+-----+----+---...---+-----+-----+
            |nciphers|sym_      |sym_      |         |sym_       |
            | = K    |cipher[0] |cipher[1] |         |cipher[K-1]|
            +--------+-----+----+-----+----+---...---+-----+-----+

             2*K + 9                    2*K + 9 + N_A_LEN
                |                          |
                v                          v
            +-------+---...---+-------+-------+---...---+-------+
            |           N_A           |           PK_A          |
            |                         |                         |
            +-------+---...---+-------+-------+---...---+-------+

                                M - 1
            +-------+---...---+-------+
            |         ignored         |
            |                         |
            +-------+---...---+-------+
```

The constant "INIT1_MAGIC" is defined in Table 1. 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. The length "N_A_LEN" and the length of "PK_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 "PK_A". When receiving "Init1", however, implementations MUST permit and ignore any bytes following "PK_A".

The "Init2" message has the following encoding:

```
    byte   0       1       2       3
        +-------+-------+-------+-------+
        |          INIT2_MAGIC          |
        |                               |
        +-------+-------+-------+-------+


            4         5       6       7       8       9
        +-------+-------+-------+-------+-------+-------+
        |          message_len          |  sym_cipher   |
        |              = M              |               |
        +-------+-------+-------+-------+-------+-------+

            10                     10 + N_B_LEN
            |                         |
            v                         v
        +-------+---...---+-------+-------+---...---+-------+
        |          N_B          |           PK_B          |
        |                       |                         |
        +-------+---...---+-------+-------+---...---+-------+

                           M - 1
        +-------+---...---+-------+
        |         ignored         |
        |                         |
        +-------+---...---+-------+
```

The constant "INIT2_MAGIC" is defined in Table 1. 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 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 "PK_B" field. When receiving "Init2", however, implementations MUST permit and ignore any bytes following "PK_B".

### 4.2. Encryption Frames

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

```
     byte   0       1       2       3              clen+2
        +-------+-------+-------+-------+---...---+-------+
        |control|      clen     |         ciphertext      |
        +-------+-------+-------+-------+---...---+-------+
```

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

The byte "control" has this structure:

```
            bit     7                 1       0
               +-------+---...---+-------+-------+
               |         cres            | rekey |
               +-------+---...---+-------+-------+
```

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:

```
        byte   0       1               plen-1
           +-------+-------+---...---+-------+
           | flags |           data          |
           +-------+-------+---...---+-------+


        byte   0       1       2       3               plen-1
           +-------+-------+-------+-------+---...---+-------+
           | flags |    urgent     |          data           |
           +-------+-------+-------+-------+---...---+-------+
```

(Note that "clen" in the previous section will generally be greater than "plen", as the ciphertext produced by the authenticated-encryption scheme must both encrypt the application data and provide a way to verify its integrity.)

The "flags" byte has this structure:

```
              bit    7    6    5    4    3    2    1    0
                   +----+----+----+----+----+----+----+----+
                   |              fres             |URGp|FINp|
                   +----+----+----+----+----+----+----+----+
```

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:

```
                    byte   0       1       2
                       +-------+-------+-------+
                       |control|     clen      |
                       +-------+-------+-------+
```

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:

```
                  byte
                     +------+------+------+------+
                   0 |      FRAME_NONCE_MAGIC    |
                     +------+------+------+------+
                   4 |                           |
                     +           offset          +
                   8 |                           |
                     +------+------+------+------+
```

The 4-byte magic constant is defined in Table 1. The 8-byte "offset" field contains an integer in big-endian format. Its value is specified in Section 3.6.

## 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 |
| 0x44415441 | FRAME_NONCE_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. 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 keys:

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

If future documents assign additional TEPs for use with tcpcrypt, they may specify different values for the lengths above. Note that the minimum session ID length required 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 [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:

```
byte   0       1       2              L - 1
    +-------+-------+-------+---...---+-------+
    |   pubkey_len  |          pubkey         |
    |      = L      |                         |
    +-------+-------+-------+---...---+-------+
```

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

Key-agreement schemes ECDHE-Curve25519 and ECDHE-Curve448 use the functions X25519 and X448, respectively, to perform the Diffie-Helman protocol as described in [RFC7748]. When using these ciphers, public-key values "PK_A" and "PK_B" are transmitted directly with no length prefix: 32 bytes for Curve25519, and 56 bytes for Curve448.

Implementations are required to implement certain TEPs, according to Table 2. Note that system administrators may configure which TEPs a host will negotiate, independent of these requirements.





| Requirement | TEP |
| --- | --- |
| MUST | TCPCRYPT_ECDHE_Curve25519 |
| SHOULD | TCPCRYPT_ECDHE_Curve448 |
| MAY | TCPCRYPT_ECDHE_P256 |
| MAY | TCPCRYPT_ECDHE_P521 |

Table 2: Requirements for implementation of TEPs

## 6. AEAD Algorithms

Specifiers and key-lengths for AEAD algorithms are given in Table 5. 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 are required to support certain algorithms according to Table 3. Note that system administrators may configure which algorithms a host will negotiate, independent of these requirements.

| Requirement | AEAD Algorithm |
|---|---|
| MUST | AEAD_AES_128_GCM |
| SHOULD | AEAD_AES_256_GCM |
| SHOULD | AEAD_CHACHA20_POLY1305 |

Table 3: Requirements for implementation of AEAD algorithms

## 7. IANA Considerations

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 the following table. 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 4.1, this document defines "sym_cipher" specifiers for which IANA is to maintain a new "tcpcrypt AEAD Algorithm" registry under the "Transmission Control Protocol (TCP) Parameters" registry, with initial values as given in the following table. The AEAD algorithms named there are defined in Section 6. Future assignments are to be made under the "RFC Required" policy detailed in [RFC8126], relying on early allocation [RFC7120] to facilitate testing before an RFC is finalized.

| Value | AEAD Algorithm | Key Length | Reference |
|---|---|---|---|
| 0x0001 | AEAD_AES_128_GCM | 16 bytes | [RFC-TBD] |
| 0x0002 | AEAD_AES_256_GCM | 32 bytes | [RFC-TBD] |
| 0x0010 | AEAD_CHACHA20_POLY1305 | 32 bytes | [RFC-TBD] |

Table 5: Authenticated-encryption algorithms corresponding to sym_cipher specifiers in Init1 and Init2 messages.

## 8. Security Considerations

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. 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.

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.

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.

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. 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 gain 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 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 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 to 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.

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 keys for past connections. All currently-specified key

agreement schemes involve ECDHE-based key agreement, meaning a new keypair can be efficiently computed for each connection. If implementations reuse these parameters, they SHOULD limit the lifetime of the private parameters as far as practical in order to minimize the number of past connections that are vulnerable.

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.

## 8.1. Asymmetric Roles

Tcpcrypt transforms a shared 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).

## 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, as this scheme is well-studied, widely implemented, and sufficiently different from the Curve25519-based scheme that it is unlikely they will both suffer from a single (non-quantum) cryptanalytic advance.

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-ecc]. Although well-trusted implementations are available as part of large cryptographic libraries, these may 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 (via SHOULD) support for key-agreement with Curve448 as this scheme appears likely to admit safe and efficient implementations; but it does not absolutely require such support, as well-proven implementations may not yet be available.

## 9. 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.

## 10. Contributors

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

## 11. References

### 11.1. 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-11 (work in progress), October 2017.

[ieee1363]
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>.

[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>.

## 11.2. 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-05 (work in progress), September 2017.

[nist-ecc]
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>.

[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