Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: January 3, 2016

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

July 2, 2015

Cryptographic protection of TCP Streams (tcpcrypt)
draft-bittau-tcpinc-tcpcrypt-03.txt

Abstract

This document presents tcpcrypt, a TCP extension for cryptographically protecting TCP connections. Tcpcrypt maintains the confidentiality of data transmitted in TCP connections against a passive eavesdropper. Additionally, applications that perform authentication can obtain end-to-end confidentiality and integrity guarantees by tying authentication to tcpcrypt Session ID values.

The extension defines a new TCP option, CRYPT, which is designed to provide compatible interworking with TCPs that do not implement tcpcrypt. The CRYPT option allows hosts to negotiate the use of tcpcrypt and establish shared, secret encryption keys. These keys are then used with an authenticated-encryption mode to protect both the confidentiality and the integrity of transmitted application data. Tcpcrypt is designed to require relatively low overhead, particularly at servers, so as to be useful even in the case of servers accepting many TCP connections per second.

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 January 3, 2016.

Table of Contents

1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Introduction

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

- Maintain confidentiality of communications against a passive adversary. Ensure that an adversary must actively intercept and modify the traffic to eavesdrop, either by re-encrypting all traffic or by forcing a downgrade to an unencrypted session.

- Permit "opportunistic" protection: that is, allow for a host configuration that employs the protocol whenever possible, without requiring the involvement of user applications.

- Provide interfaces to higher-level software to facilitate end-to-end security. For example, exposing a session ID to an application would allow it to compare this value with its peer (after the fact, or via some real-time mechanism) and thereby detect any active modification of the connection.

- Minimize computational cost, particularly on servers.

- Operate independently of IP addresses, making it possible to authenticate resumed TCP connections even when either end changes IP address.

- Facilitate multipath TCP [RFC6824] by identifying a TCP stream with a session ID independent of IP addresses and port numbers.

- Provide for incremental deployment and graceful fallback, even in the presence of NATs and other middleboxes that might remove unknown options, and traffic normalizers.

3. Idealized protocol

This section describes the tcpcrypt protocol at an abstract level, without reference to particular cryptographic algorithms or data encodings. Readers who simply wish to see the extention-negotiation and key-exchange protocols should skip to Section 3.4.

3.1. Stages of the protocol

A tcpcrypt endpoint goes through multiple stages. It begins in a setup phase and ends up in one of two states, ENCRYPTING or DISABLED, before applications may send or receive data. The ENCRYPTING and DISABLED states are definitive and mutually exclusive; an endpoint that has been in one of the two states MUST NOT ever enter the other, nor ever re-enter the setup phase.

3.1.1. The setup phase

In the setup phase, two hosts negotiate use of the tcpcrypt extension and then, if it will be used, agree on a suite of cryptographic algorithms and establish secret session keys.

After establishing that both hosts are willing to perform the tcpcrypt protocol, the setup phase uses the Data portion of TCP segments to exchange cryptographic keys. Implementations MUST NOT include application data in TCP segments during setup and MUST NOT allow applications to read or write data. System calls MUST behave the same as for TCP connections that have not yet entered the ESTABLISHED state; calls to read and write SHOULD block or return temporary errors, while calls to poll or select SHOULD consider connections not ready.

When setup succeeds, tcpcrypt enters the ENCRYPTING state. A successful setup also produces an important value called the _Session ID_. The Session ID is tied to the negotiated algorithms and cryptographic keys, and is unique over all time with overwhelming probability.

Operating systems MUST make the Session ID available to applications. To prevent man-in-the-middle attacks, applications MAY authenticate the session ID through any protocol that ensures both endpoints of a connection have the same value. Alternatively, Applications MAY simply log Session IDs so as to enable attack detection after the fact, by comparing the values logged at each end.

The setup phase can also fail for various reasons, in which case tcpcrypt enters the DISABLED state.

Applications MAY test whether setup succeeded by querying the operating system for the Session ID. Requests for the Session ID MUST return an error when tcpcrypt is not in the ENCRYPTING state. Applications SHOULD authenticate the returned Session ID. Applications relying on tcpcrypt for security SHOULD authenticate the Session ID and SHOULD treat unauthenticated Session IDs the same as connections in the DISABLED state.

3.1.2. The ENCRYPTING state

When the setup phase succeeds, tcpcrypt enters the ENCRYPTING state. Once in this state, applications may read and write data with the expected semantics of TCP connections.

In the ENCRYPTING state, a host MUST employ the authenticated-encryption protocol described in Section 4.4.4 to protect all transmitted application data.

Once it has entered the ENCRYPTING state, an endpoint MUST NOT ever transition, directly or indirectly, to the DISABLED state.

3.1.3. The DISABLED state

When setup fails, tcpcrypt enters the DISABLED state. In this case, the host MUST continue just as TCP would without tcpcrypt, unless network conditions would cause a plain TCP connection to fail as well. Entering the DISABLED state prohibits the endpoint from ever entering the ENCRYPTING state.

An implementation MUST behave identically to ordinary TCP in the DISABLED state, except that the first segment transmitted after entering the DISABLED state MAY include a TCP CRYPT option with a DECLINE suboption (and optionally other suboptions such as UNKNOWN) to indicate that tcpcrypt is supported but not enabled. Section 4.3.2 describes how this is done.

Operating systems MUST allow applications to turn off tcpcrypt by setting the state to DISABLED before opening a connection. An active opener with tcpcrypt disabled MUST behave identically to an implementation of TCP without tcpcrypt. A passive opener with tcpcrypt disabled MUST also behave like normal TCP, except that it MAY optionally respond to SYN segments containing a CRYPT option with SYN-ACK segments containing a DECLINE suboption, so as to indicate that tcpcrypt is supported but not enabled.

3.2. Cryptographic algorithms

The setup phase employs three types of cryptographic algorithms:

- A _public-key cipher_ is used with a short-lived public key to exchange (or agree upon) a random, shared secret.

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

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

The Extract and CPRF functions used by default are the Extract and Expand functions of HKDF [RFC5869]. These are defined as follows:

```
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) | ...
```

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

Because the public key cipher, the extract function, and the expand function all make use of cryptographic hashes in their constructions, the three algorithms are negotiated as a unit employing a single hash function. For example, the OAEP+-RSA [RFC2437] cipher, which uses a SHA-256-based mask-generation function, is coupled with HKDF [RFC5869] using HMAC-SHA256 [RFC2104].

The ENCRYPTING phase employs an _authenticated encryption mode_ to encrypt and integrity-protect all application data.

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

3.3. "C" and "S" roles

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

We use the terms "C" and "S" to denote the distinct roles of the two hosts in tcpcrypt's setup phase. In the case of key transport, "C" is the host that supplies a public key, while "S" is the host that encrypts a pre-master secret with the key belonging to "C". Which role a host plays can have performance implications, because for some public key algorithms encryption is much faster than decryption. For instance, on a machine at the time of writing, encryption with a 2,048-bit RSA-3 key is over two orders of magnitude faster than decryption.

Because servers often need to establish connections at a faster rate than clients, and because servers are often passive openers, by default the passive opener plays the "S" role. However, implementations MUST provide a mechanism for the passive opener to reverse roles and play the "C" role, as discussed in Section 4.2.

3.4. Protocol negotiation

When a host C connects to S, the two use the TCP option TCPCRYPT to negotiate the use of the tcpcrypt extension. If the server is willing, the following exchange will occur:

```
C -> S:  HELLO
S -> C:  PKCONF, pub-cipher-list
```

The pub-cipher-list value is a list of public-key ciphers and parameters acceptable to S. These are defined in Figure 2.

If S is not willing to engage in tcpcrypt, it may respond with DECLINE instead; if it does not implement tcpcrypt, it will fail to respond to the HELLO at all. In either case, C will immediately transition to the DISABLED state.

3.5. Key exchange protocol

If the protocol negotiation above indicates that both sides wish to employ tcpcrypt, they will then use the TCP datastream to continue with this key-exchange protocol:

```
C -> S:  INIT1, sym-cipher-list, N_C, pub-cipher, PK_C
S -> C:  INIT2, sym-cipher, KX_S
```

The parameters are defined as follows:

- sym-cipher-list: a list of symmetric ciphers (AEAD algorithms) acceptable to C. These are specified in Table 6.

- N_C: a nonce chosen at random by C.

- pub-cipher: a choice from the pub-cipher-list received in protocol negotiation. Determines the type of PK_C.

- PK_C: C's public key or key agreement parameter. This is a short-lived value that SHOULD be refreshed periodically and SHOULD NOT ever be written to persistent storage.

- sym-cipher: the symmetric cipher (AEAD algorithm) selected by S.

- KX_S: key-exchange information produced by S. KX_S will depend on whether key transport is being done (e.g., RSA) or key agreement (e.g., Diffie-Hellman). KX_S is defined in Table 1.

Cipher	KX_S	PMS
OAEP+-RSA exp3	ENC (PK_C, R_S)	R_S
ECDHE	N_S, PK_S	key-agreement-output

ENC (PK_C, R_S) denotes an encryption of R_S with public key PK_C. R_S and N_S are random values chosen by S. Their lengths are defined in Figure 2. PK_S is S's key agreement parameter. PMS is the Pre Master Secret from which keys are ultimately derived.

Table 1

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

```
param  := { num-pub-ciphers, pub-cipher-list, init1, init2 }
PRK    := Extract (N_C, { param, PMS })
```

Here num-pub-ciphers is a single octet specifying how many three-byte algorithm specifiers were provided by the "S" host in a PKCONF suboption (described in Section 4.3.4). pub-cipher-list is this many three-byte specifiers, taken from the body of the PKCONF suboption.

init1 and init2 are the encodings of the Init1 and Init2 messages described in Section 4.4.3.

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

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

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

The value ss[0] is used to generate all key material for the current connection. SID[0] is the session ID for the current connection, and will with overwhelming probability be unique for each individual TCP connection. The most computationally expensive part of the key exchange protocol is the public key cipher. The values of ss[i] for i > 0 can be used to avoid public key cryptography when establishing subsequent connections between the same two hosts, as described in Section 3.8. The CONST values are constants defined in Table 7. The K_LEN values and nonce sizes are negotiated, and are specified in Figure 2.

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

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

Where flags is a single octet from 0x0 to 0x3, computed as follows:

```
bit  0 1 2 3 4 5 6 7
    +-+-+-+-+-+-+-+-+
    |0 0 0 0 0 0 s c|
    +-+-+-+-+-+-+-+-+
```

Here bit "s" is set when the "S" mode host has indicated application-level support for tcpcrypt. The "c" bit is set when the "C" mode host has indicated application-level support for tcpcrypt. Both bits are 0 by default unless the application has enabled the TCP_CRYPT_SUPPORT option described in Section 6.

Finally, each master key mk is used to generate keys for authenticated encryption for the "S" and "C" roles. Key k_cs is used by "C" to encrypt and "S" to decrypt, while k_sc is used by "S" to encrypt and "C" to decrypt.

```
k_cs := CPRF (mk, CONST_KEY_C, ae_keylen)
k_sc := CPRF (mk, CONST_KEY_S, ae_keylen)
```

The ae_keylen depends on the authenticated-encryption algorithm selected, and is given under "Key Length" in Table 6.

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

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

3.6. Data encryption and authentication

Following key exchange, all further communication in a tcpcrypt-enabled connection is carried out within delimited data-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 6. One algorithm is selected during the negotiation described in Section 3.5.

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_cs or k_sc, according to the host's role as described in Section 3.5.

Further, tcpcrypt provides inputs of the following types to the authenticated encryption operation, where the datatypes and their values are defined in Section 4.4.4:

N: FrameNonce
A: AssocData
P: PlainText

The output of the encryption operation, C, is transmitted as the frame's "ciphertext" value.

When a frame is received, tcpcrypt reconstructs the N and A values from data sent in the clear, and provides these and the ciphertext value to the the authenticated decryption operation. The output of this operation is either P, a value of type "Plaintext", or the

special symbol FAIL. In the latter case, the implementation may either ignore the frame or terminate the connection.

3.7. Re-keying

We refer to the two encryption keys (k_cs, k_sc) as a _key set_. We refer to the key set generated by mk[i] as the key set with _generation number_ i within a session. Initially, the two hosts use the key set with generation number 0.

Either host may decide to evolve the encryption key at one or more points within a session, by incrementing the generation number of its transmit keys. When switching keys to generation j, a host must label the frames it transmits with a REKEY message containing j, so that the recipient host knows to decrypt the application data using the new keyset.

Upon receiving a REKEY<j> message, a recipient using transmit keys from a generation less than j must also update its transmit keys and start including a REKEY<j> message in its outgoing frames. A host must continue transmitting REKEY messages until its peer acknowledges the switch to the new keys.

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.

3.8. Session caching

When two hosts have already negotiated session secret ss[i-1], they can establish a new connection without public key operations using ss[i]. The four-message protocol of Section 3.5 is replaced by:

```
A -> B:  NEXTK1, SID[i]
B -> A:  NEXTK2
```

Which symmetric keys a host uses for transmitted segments is determined by its role in the original session ss[0]. It does not depend on which host is the passive opener in the current session. If A had the "C" role in the first session, then A uses k_cs for sending segments and k_sc for receiving. Otherwise, if A had the "S" role originally, it uses k_sc and k_cs, respectively. B similarly uses the transmit keys that correspond to its role in the original session.

After using ss[i] to compute mk[0], implementations SHOULD compute and cache ss[i+1] for possible use by a later session, then erase ss[i] from memory. Hosts SHOULD keep ss[i+1] around for a period of

time until it is used or the memory needs to be reclaimed. Hosts SHOULD NOT write a cached ss[i+1] value to non-volatile storage.

It is an implementation-specific issue as to how long ss[i+1] should be retained if it is unused. If the passive opener times it out before the active opener does, the only cost is the additional ten bytes to send NEXTK1 for the next connection. The behavior then falls back to a normal public-key handshake.

3.8.1. Session caching control

Implementations MUST allow applications to control session caching by setting the following option:

TCP_CRYPT_CACHE_FLUSH When set on a TCP endpoint that is in the ENCRYPTING state, this option causes the operating system to flush from memory the cached ss[i+1] (or ss[i+1+n] if other connections have already been established). When set on an endpoint that is in the setup phase, causes any cached ss[i] that would have been used to be flushed from memory. In either case, future connections will have to undertake another round of the public key protocol in Section 3.5. Applications SHOULD set TCP_CRYPT_CACHE_FLUSH whenever authentication of the session ID fails.

4. Extensions to TCP

The tcpcrypt extension adds a new kind of option, CRYPT. During the setup phase, all TCP segments MUST have the CRYPT option.

The idealized protocol of the previous section is performed partly with TCP options during the handshake, and partly in the Data portion of TCP segments (after the SYN exchanges). In particular, the negotiation of the tcpcrypt extension is embedded in the handshake, and successful negotiation then allows all further communications to be framed in the TCP datastream: the INIT1 and INIT2 messages accomplish the exchange of keys, and are followed by encrypted application data.

4.1. Protocol states

The setup phase is divided into six states: CLOSED, NEXTK-SENT, HELLO-SENT, C-MODE, LISTEN, and S-MODE. Together with the ENCRYPTING and DISABLED states already discussed, this means a tcpcrypt endpoint can be in one of eight states.

In addition to tcpcrypt's state, each endpoint will also be in one of the 11 TCP states described in the TCP protocol specification

[RFC0793]. Not all pairs of states are valid. Table 2 shows which TCP states an endpoint can be in for each tcpcrypt state.

Tcpcrypt state	TCP states for an active opener	TCP states for a passive opener
CLOSED	CLOSED	CLOSED
NEXTK-SENT	SYN-SENT	n/a
HELLO-SENT	SYN-SENT	SYN-RCVD
C-MODE	ESTABLISHED, FIN-WAIT-1	ESTABLISHED, FIN-WAIT-1
LISTEN	n/a	LISTEN
S-MODE	(SYN-RCVD), ESTABLISHED	SYN-RCVD
ENCRYPTING	(SYN-RCVD), ESTABLISHED+	SYN-RCVD, ESTABLISHED+
DISABLED	any	any

Valid tcpcrypt and TCP state combinations. States in parentheses occur only with simultaneous open. ESTABLISHED+ means ESTABLISHED or any later state (FIN-WAIT-1, FIN-WAIT-2, CLOSING, TIME-WAIT, CLOSE-WAIT, or LAST-ACK).

Table 2

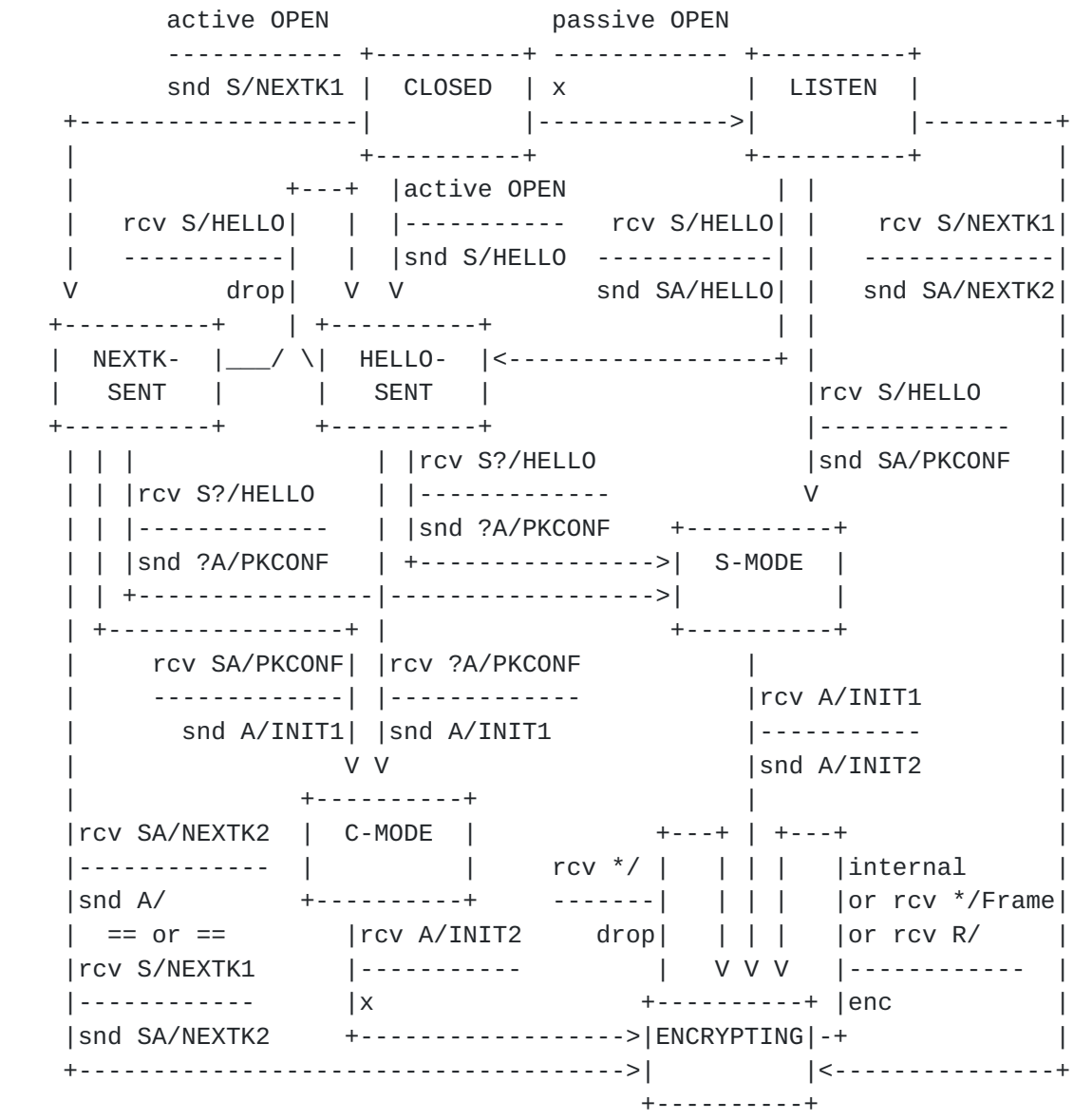
Figure 1 shows how tcpcrypt transitions between states. Each transition is labeled by events that may trigger the transition above the line, and an action the local host is permitted to take in response below the line. "snd" and "rcv" denote sending and receiving segments, respectively. "internal" means any possible event except for receiving a segment (i.e., timers and system calls). "drop" means discarding the last received segment and preventing it from having any effect on TCP's state. "enc" means any valid TCP action, including no action, except that any segments transmitted must contain encrypted frames of application data. "x" indicates that a host sends no segments when taking a transition.

A segment is described as "F/Op". F specifies constraints on the control bits of the TCP header, as follows:

F	Meaning
S	SYN=1, ACK=0, FIN=0, RST=0
SA	SYN=1, ACK=1, FIN=0, RST=0
A	SYN=0, ACK=1, FIN=0, RST=0
S?	SYN=1, ACK=any, FIN=0, RST=0
?A	SYN=any, ACK=1, FIN=0, RST=0
R	RST=1
*	any

Op designates message types in the abstract protocol, which also correspond to particular suboptions of the TCP CRYPT option, described in Section 4.3; or if Op is "Frame" it refers to a data portion that either contains successfully-authenticated framed data or else contains incomplete frames which must be buffered before further processing. A segment with SYN=1 and ACK=0 that contains the NEXTK1 suboption will also explicitly or implicitly contain the HELLO suboption; such a segment matches event constraints on either option--e.g., it matches any of the "rcv S/HELLO", "rcv S?/HELLO", and "rcv S/NEXTK1" events. An empty Op matches any segment with the appropriate control bits.

The "drop" transitions from NEXTK-SENT and HELLO-SENT to HELLO-SENT change TCP slightly by ignoring a segment and preventing a TCP transition from SYN-SENT to SYN-RCVD that would otherwise occur during simultaneous open. Therefore, these transitions SHOULD be disabled by default. They MAY be enabled on one side by an application that wishes to enable tcpcrypt on simultaneous open, as discussed in Section 4.2.1.



State diagram for tcpcrypt. Transitions to DISABLED and CLOSED are not shown.

Figure 1

Any segment that would be discarded by TCP (e.g., for being out of window) MUST also be ignored by tcpcrypt. However, certain segments that might otherwise be accepted by TCP MUST be dropped by tcpcrypt and prevented from affecting TCP's state.

Except for these drop actions, tcpcrypt MUST abide by the TCP protocol specification [RFC0793]. Thus, any segment transmitted by a host MUST be permitted by the TCP specification in addition to matching either a transition in Figure 1 or one of the transitions to

DISABLED or CLOSED described below. In particular, a host MUST NOT acknowledge an INIT1 segment unless either the acknowledgment contains an INIT2 or the host transitions to DISABLED.

Various events cause transitions to DISABLED from states other than ENCRYPTING. In particular:

- Operating systems MUST provide a mechanism for applications to transition to DISABLED from the CLOSED and LISTEN states.

- A host in the setup phase MUST transition to DISABLED upon receiving any segment without a TCP CRYPT option.

- A host in the setup phase MUST transition to DISABLED upon receiving any segment with the FIN or RST control bit set.

- A host in the setup phase MUST transition to DISABLED upon sending a segment with the FIN bit set. (As discussed below, however, a host MUST NOT send a FIN segment from the C-MODE state.)

Other specific conditions cause a transition to DISABLED and are discussed in the sections that follow.

CLOSED is a pseudo-state representing a connection that does not exist. A tcpcrypt connection's lifetime is identical to that of its associated TCP connection. Thus, tcpcrypt transitions to CLOSED exactly when TCP transitions to CLOSED.

The only valid tcpcrypt state transition from ENCRYPTING is to CLOSED, which occurs only when TCP transitions to CLOSED. tcpcrypt per se cannot cause TCP to transition to CLOSED.

If a CLOSE happens in the ENCRYPTING state, a host MUST send the ApplicationFin message before transitioning to CLOSED.

A host MUST NOT send a FIN segment from the C-MODE state. The reason is that the remote side can be in the ENCRYPTING state and would thus require receipt of an ApplicationFin message to securely signal the end of data, yet a host in C-MODE cannot compute the necessary encryption keys before receiving the INIT2 segment.

If a CLOSE happens in C-MODE, a host MUST delay sending a FIN segment until receiving an ACK for its INIT1 segment. If the remote host is in ENCRYPTING, the ACK segment will contain INIT2 and the local host can transition to ENCRYPTING before sending ApplicationFin along with the FIN flag. If the remote host is not in ENCRYPTING, the ACK will not contain INIT2, and thus the local host can transition to DISABLED before sending the FIN flag.

If a CLOSE happens in C-MODE, an implementation MAY delay processing the CLOSE event and entering the TCP FIN-WAIT-1 state until sending the FIN. If it does not, the implementation MUST ensure all relevant timers correspond to the time of transmission of the FIN segment, not the time of entry into the FIN-WAIT-1 state.

4.2. Role negotiation

A passive opener receiving an S/HELLO segment may choose to play the "S" role (by transitioning to S-MODE) or the "C" role (by transitioning to HELLO-SENT). An active opener may accept the role not chosen by the passive opener, or may instead disable tcpcrypt. During simultaneous open, one endpoint must choose the "C" role while the other chooses the "S" role. Operating systems MUST allow applications to guide these choices on a per-connection basis.

Applications SHOULD be able to exert this control by setting a per-connection _CMODE disposition_, which can take on one of the following five values:

TCP_CRYPT_CMODE_DEFAULT This disposition SHOULD be the default. A passive opener will only play the "S" role, but an active opener can play either the "C" or the "S" role. Simultaneous open without session caching will cause tcpcrypt to be disabled unless the remote host has set the TCP_CMODE_ALWAYS[_NK] disposition.

TCP_CRYPT_CMODE_ALWAYS

TCP_CRYPT_CMODE_ALWAYS_NK With this disposition, a host will only play the "C" role. The _NK version additionally prevents the use of session caching if the session was originally established in the "S" role.

TCP_CRYPT_CMODE_NEVER

TCP_CRYPT_CMODE_NEVER_NK With this disposition, a host will only play the "S" role. The _NK version additionally prevents the use of session caching if the session was originally established in the "C" role.

The CMODE disposition prohibits certain state transitions, as summarized in Table 3. If an event occurs for which all valid transitions in Figure 1 are prohibited, a host MUST transition to DISABLED. Operating systems MAY add additional CMODE dispositions, for instance to force or prohibit session caching.

CMODE disposition	Prohibited transitions
TCP_CRYPT_CMODE_DEFAULT	LISTEN --> HELLO-SENT
	HELLO-SENT --> HELLO-SENT
	NEXTK-SENT --> HELLO-SENT
TCP_CRYPT_CMODE_ALWAYS[_NK]	any --> S-MODE
TCP_CRYPT_CMODE_NEVER[_NK]	LISTEN --> HELLO-SENT
	HELLO-SENT --> HELLO-SENT
	NEXTK-SENT --> HELLO-SENT
	any --> C-MODE

State transitions prohibited by each CMODE disposition

Table 3

4.2.1. Simultaneous open

During simultaneous open, two ends of a TCP connection are both active openers. If both hosts attempt to use session caching by simultaneously transmitting S/NEXTK1 segments, and if both transmit the same session ID, then both may reply with SA/NEXTK2 segments and immediately enter the ENCRYPTING state. In this case, the host that played "C" when the session was initially negotiated MUST use the symmetric encryption keys for "C" (i.e., encrypt with k_cs, decrypt with k_sc), while the host that initially played "S" uses the "S" keys for the new connection.

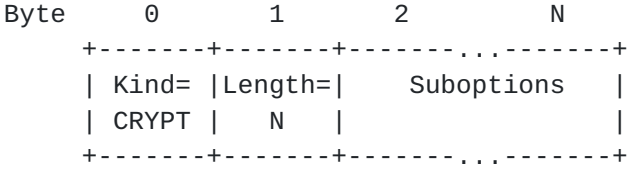
If both hosts in a simultaneous open do not attempt to use session caching, or if the two hosts use incompatible Session IDs, then they MUST engage in public-key-based key negotiation to use tcpcrypt. Doing so requires one host to play the "C" role and the other to play the "S" role. With the TCP_CRYPT_CMODE_DEFAULT disposition, these roles are usually determined by the passive opener choosing the "S" role. With no passive opener, both active openers will end up in S-MODE, then transition to DISABLED upon receiving an unexpected PKCONF.

Simultaneous open can work with key negotiation if exactly one of the two hosts selects the TCP_CRYPT_CMODE_ALWAYS disposition. This host will then drop S/HELLO segments and remain in C-MODE while the other host transitions to S-MODE. Applications SHOULD NOT set TCP_CRYPT_CMODE_ALWAYS on both sides of a simultaneous open, as this will result in tcpcrypt being disabled. The reception of two

simultaneous HELLO (or NEXTK) messages will disable tcpcrypt because it is not explicit as to who is playing the "C" or "S" role.

4.3. The TCP CRYPT option

A CRYPT option has the following format:

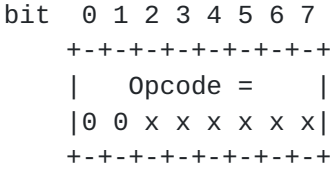


Format of TCP CRYPT option

Kind is always CRYPT. Length is the total length of the option, including the two bytes used for Kind and Length. These first two bytes are then followed by zero or more suboptions. Suboptions determine the meaning of the TCP CRYPT option. When a TCP header contains more than one CRYPT option, a host MUST interpret them the same as if all the suboptions appeared in a single CRYPT option. This makes tcpcrypt options future-proof as new suboptions can be placed in a separate CRYPT option, which can be ignored if not understood, while other CRYPT options can still be processed.

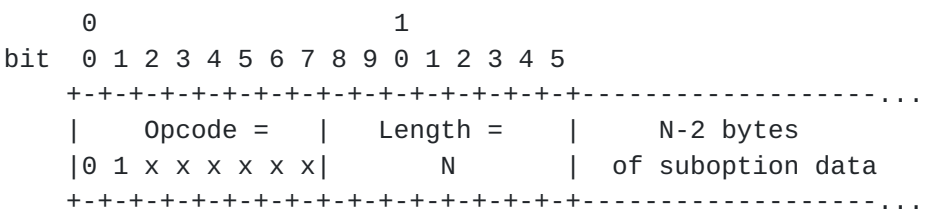
Each suboption begins with an Opcode byte. The specific format of the option depends on the two most significant bits of the Opcode.

Suboptions with opcodes from 0x00 to 0x3f contain no data other than the single opcode byte:



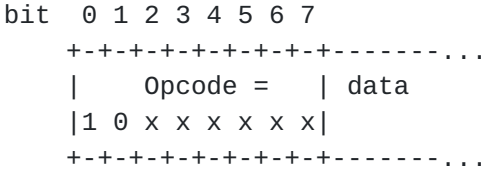
Hosts MUST ignore any opcodes of this format that they do not recognize.

Suboptions with opcodes from 0x40 to 0x7f contain an opcode, a length field, and data bytes.



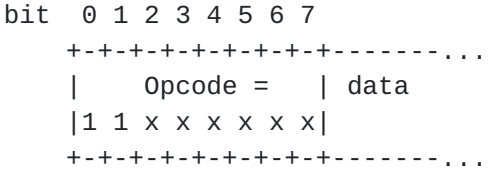
Hosts MUST ignore any opcodes of this format that they do not recognize.

Suboptions with opcodes from 0x80 to 0xbf contain zero or more bytes of data whose length depends on the opcode. These suboptions can be either fixed length or variable length; implementations that understand these opcodes will known which they are; if the suboption is fixed length the implementation will know the length; otherwise it will know where to look for the length field.



If a host sees an unknown opcode in this range, it MUST ignore the suboption and all subsequent suboptions in the same TCP CRYPT option. However, if more than one CRYPT option appears in the TCP header, the host MUST continue processing suboptions from the next TCP CRYPT option. Skipping suboptions in the TCP CRYPT option applies only to this option range since the length of the suboption cannot be determined by the receiver. In other cases, where the length is known, the receiver skips to the next suboption.

Suboptions with opcodes from 0xc0 to 0xff also contain an opcode-specific length of data. As before, these suboptions can be either fixed length or variable length. Suboptions in this range are classed as mandatory as far as the protocol is concerned. However, they are not MANDATORY to implement unless otherwise stated, as discussed below.



Should a host encounter an unknown opcode greater than or equal to 0xc0 during the setup phase of the protocol, the host MUST transition to the DISABLED state. It SHOULD respond with both a DECLINE

suboption and an UNKNOWN suboption specifying the opcode of the unknown mandatory suboption, after which the host MUST NOT send any further CRYPT options.

Should a host encounter an unknown opcode greater than or equal to 0xc0 while in the ENCRYPTING state, the host MUST respond with an UNKNOWN suboption specifying the opcode of the unknown mandatory suboption, and should ensure the session continues with the same encryption and authentication state as it had before the segment was received. This may require ignoring other suboptions within the same message, or reverting any half-negotiated state.

Table 4 summarizes the opcodes discussed in this document. It is MANDATORY that all implementations support every opcode in this table. Each opcode is listed with the length in bytes of the suboption (including the opcode byte), or * for variable-length suboptions. The last column specifies in which of the (S)etup phase, (E)NCRYPTING state, and (D)ISABLED state an opcode may be used. A host MUST NOT send an option unless it is in one of the stages indicated by this column.

Value	Length	Name	Stages
0x01	1	HELLO	S
0x02	1	HELLO-app-support	S
0x03	1	HELLO-app-mandatory	S
0x04	1	DECLINE	SD
0x05	1	NEXTK2	S
0x06	1	NEXTK2-app-support	S
0x41	*	PKCONF	S
0x42	*	PKCONF-app-support	S
0x43	*	UNKNOWN	SED
0x44	*	SYNCOOKIE	S
0x45	*	ACKCOOKIE	SED
0x84	10	NEXTK1	S

Opcodes for suboptions of the TCP CRYPT option.

Table 4

If a TCP segment (sent by an active opener) has the SYN flag set, the ACK flag clear, and one or more TCP CRYPT options, there is an implicit HELLO suboption even if that suboption does not appear in the segment. In particular, when such a SYN segment contains a single, empty, two-byte TCP CRYPT option, the passive opener MUST interpret that option as equivalent to the three-byte TCP option

composed of bytes CRYPT, 3, 1 (Kind = CRYPT, Length = 3, Suboption = HELLO).

A host MUST enter the DISABLED state if, during the setup phase, it receives a segment containing no TCP CRYPT option. This is for robustness against middleboxes that strip options. A host MUST also enter DISABLED if, during the setup phase, it receives a DECLINE suboption or any unrecognized suboption with opcode greater than or equal to 0xc0. The DECLINE option is the preferred way for a host to refuse tcpcrypt. A host MAY also choose to reply without a TCP CRYPT option to disable tcpcrypt. Once a host has entered DISABLED, the host MAY include a CRYPT option in the next segment transmitted, but only if the segment also contains the DECLINE suboption. All subsequently transmitted packets MUST NOT contain the CRYPT option.

We now precisely specify the format of each suboption. In the sections that follow, all multi-byte values are encoded in big-endian format.

4.3.1. The HELLO suboption

The HELLO dataless suboption MUST only appear in a segment with the SYN control bit set. It is used by an active opener to indicate interest in using tcpcrypt for a connection, and by a passive opener to indicate that the passive opener wishes to play the "C" role.

The initial SYN segment from an active opener wishing to use tcpcrypt MUST contain a TCP CRYPT option with either an explicit or an implicit HELLO suboption.

After receiving a SYN segment with the HELLO suboption, a passive opener MUST respond in one of three ways:

- To continue setting up tcpcrypt and play the "S" role, the passive opener MUST respond with a PKCONF suboption in the SYN-ACK segment and transition to S-MODE.

- To continue setting up tcpcrypt and play the "C" role, the passive opener MUST respond with a HELLO suboption in the SYN-ACK segment and transition to HELLO-SENT.

- To continue without tcpcrypt, the passive opener MUST respond with either no CRYPT option or the DECLINE suboption in the SYN-ACK segment, then transition to the DISABLED state.

An active opener receiving HELLO in a SYN-ACK segment must either transition to S-MODE and respond with a PKCONF suboption, or transition to DISABLED.

There are three variants of the HELLO option used for application-level authentication, each encoded differently as shown in Table 4. The variants are: a plain HELLO where the application is not tcpcrypt-aware (but the kernel is), an "application supported" HELLO where the application is tcpcrypt-aware and is advertising the fact, and a "application mandatory" HELLO where the application requires the remote application to support tcpcrypt otherwise the connection MUST revert to plain TCP. The application supported HELLO can be used, for example, when implementing HTTP digest authentication - an application can check whether the peer's application is tcpcrypt aware and proceed to authenticate tcpcrypt's session ID over HTTP, otherwise reverting to standard HTTP digest authentication. The application mandatory HELLO can be used, for example, when implementing an SSL library that attempts tcpcrypt but reverts to SSL if the peer's SSL library does not support tcpcrypt. The application mandatory HELLO avoids double encrypting (SSL-over-tcpcrypt) since the connection will revert to plain TCP if the remote SSL library is not tcpcrypt-aware.

4.3.2. The DECLINE suboption

The DECLINE dataless suboption is sent by a host to indicate that the host will not enable tcpcrypt on a connection. If a host is in the DISABLED state or transitioning to the DISABLED state, and the host transmits a segment containing a CRYPT option, then the segment MUST contain the DECLINE suboption.

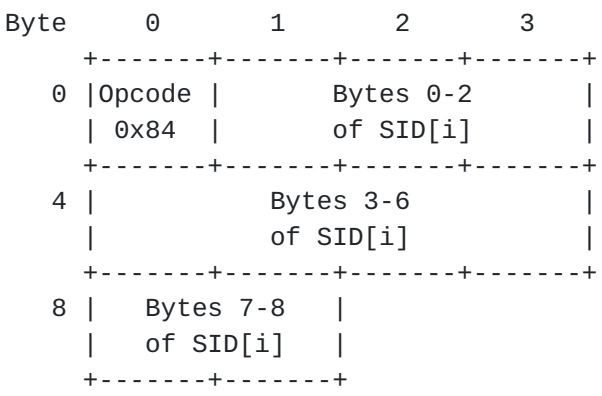
A passive opener SHOULD send a DECLINE suboption in response to a HELLO suboption or NEXTK1 suboption in a received SYN segment if it supports tcpcrypt but does not wish to engage in encryption for this particular session.

Implementations MUST NOT send segments containing the DECLINE suboption from the C-MODE or ENCRYPTING states.

4.3.3. The NEXTK1 and NEXTK2 suboptions

The NEXTK1 suboption MUST only appear in a segment with the SYN control bit set and the ACK bit clear. It is used by the active opener to initiate a TCP session without the overhead of public key cryptography. The new session key is derived from a previously negotiated session secret, as described in Section 3.8.

The suboption is always 10 bytes in length; the data contains the first nine bytes of SID[i] and is used to to start the session with session secret ss[i]. The format of the suboption is:



Format of the NEXTK1 suboption

The active opener MUST use the lowest value of i that has not already appeared in a NEXTK1 segment exchanged with the same host and for the same pre-session seed.

If the passive opener recognizes SID[i] and knows ss[i], it SHOULD respond with a segment containing the dataless NEXTK2 suboption. The NEXTK2 option MUST only appear in a segment with both the SYN and ACK bits set.

If the passive opener does not recognize SID[i], or SID[i] is not valid or has already been used, the passive opener SHOULD respond with a PKCONF or HELLO option and continue key negotiation as usual.

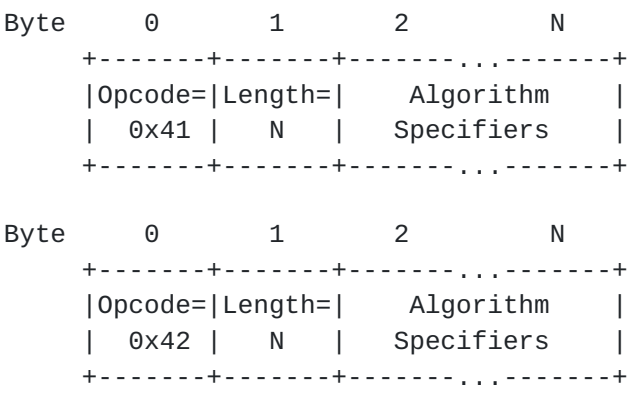
When two hosts have previously negotiated a tcpcrypt session, either host may use the NEXTK1 option regardless of which host was the active opener or played the "C" role in the previous session. However, a given host must either encrypt with k_cs for all sessions derived from the same pre-session seed, or k_sc. Thus, which keys a host uses to send segments depends only whether the host played the "C" or "S" role in the initial session that used ss[0]; it is not affected by which host was the active opener transmitting the SYN segment containing a NEXTK1 suboption.

A host MUST reject a NEXTK1 message if it has previously sent or received one with the same SID[i]. In the event that two hosts simultaneously send SYN segments to each other with the same SID[i], but the two segments are not part of a simultaneous open, both connections will have to revert to public key cryptography. To avoid this limitation, implementations MAY choose to implement session caching such that a given pre-session key is only good for either passive or active opens at the same host, not both.

In the case of simultaneous open, two hosts that simultaneously send SYN packets with NEXTK1 and the same SID[i] may establish a connection, as described in Section 4.2.1.

4.3.4. The PKCONF suboption

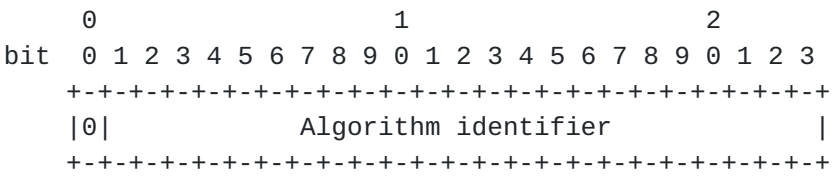
The PKCONF option has one of the following two formats:



Formats of the PKCONF suboption

The two are treated identically by tcpcrypt, except that opcode 0x42 (PKCONF-app-support) signals that the application on the sending host has set the TCP_CRYPT_SUPPORT option to non-zero, and hence the receiving host should return 1 for the TCP_CRYPT_PEER_SUPPORT socket option, as discussed in Section 6.

The suboption data, whose length (N-2) must be divisible by 3, contains one or more 3-byte algorithm specifiers of the following form:



Format of algorithm specifier within PKCONF. Fields starting with 1 are reserved for future use by algorithm identifiers longer than three bytes.

The algorithm identifier specifies a number of parameters, defined in Figure 2.

Hosts MUST implement OAEP+-RSA3 and ECDHE-P256 and ECDHE-P521, but MAY by default disable certain algorithms and key sizes. In particular, implementations SHOULD disable larger RSA keys (Algorithm

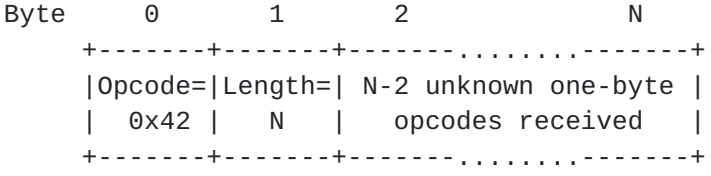
identifiers 0x102-0x103) by default unless such larger keys and ciphertexts can fit into a single TCP segment.

Servers demanding utmost performance SHOULD use RSA because the RSA encrypt operation is must faster than Diffie-Hellman operations, resulting in a higher connection rate.

Depending on the encoding of the PKCONF suboption (see Table 4), it can indicate whether "S's" application is tcpcrypt-aware or not. For the "C" role, the encoding of the HELLO suboption does this. This mechanism can be used for bootstrapping application-level authentication without requiring probing in upper layer protocols to check for support (which may not be possible). The application controls these encodings via the TCP_CRYPT_SUPPORT socket option.

4.3.5. The UNKNOWN suboption

The UNKNOWN option has the following format:



Format of the UNKNOWN suboption

This suboption is sent in response to an unrecognized suboption that has been received. The contents of the option are a complete list of the mandatory suboption opcodes from the received packet that were not understood. Note that this option is only sent once, in the next packet that the host sends. This means that it is reliable when sent in a SYN-ACK, but unreliable otherwise. Any mechanism sending new mandatory attributes must take this into account. If multiple packets, each containing unrecognized options, are received before an UNKNOWN suboption can be sent, the options list MUST contain the union of the two sets. The order of the opcode list is not significant.

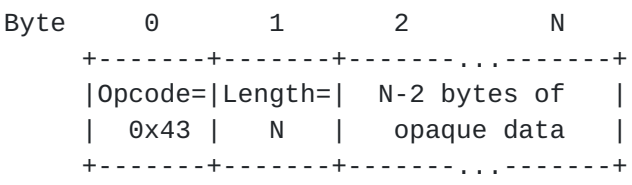
If a host receives an unrecognized option, it SHOULD reply with the UNKNOWN suboption to notify the other side. If the host transitions to DISABLED as a result of the unrecognized option, then the host MUST also include the DECLINE suboption if it sends an UNKNOWN suboption (or more generally if it includes a CRYPT option in the next packet).

As a special case, if PKCONF (0x41) or INIT1 (0x06) appears in the unrecognized opcode list, it does not mean the sender does not

understand the option (since these options are MANDATORY). Instead, it means the sender does not implement any of the algorithms specified in the PKCONF or INIT1 message. In either case, the segment must also contain a DECLINE suboption.

4.3.6. The SYNCOOKIE and ACKCOOKIE suboptions

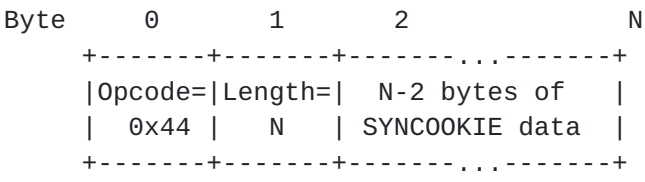
A passive opener MAY include the SYNCOOKIE suboption in a segment with both the SYN and ACK flags set. SYNCOOKIE allows a server to be stateless until the TCP handshake has completed. It has the following format:



Format of the SYNCOOKIE suboption

The data is opaque as far as the protocol is concerned; it is entirely up to implementations how to make use of this suboption to hold state. It is OPTIONAL to send a SYNCOOKIE, but MANDATORY to understand and respond to them.

The ACKCOOKIE suboption echoes the contents of a SYNCOOKIE; it MUST be sent in a packet acknowledging receipt of a packet containing a SYNCOOKIE, and MUST NOT be sent in any other packet. It has the following format:



Format of the ACKCOOKIE suboption

Servers that rely on suboption data from ACKCOOKIE to reconstruct session state SHOULD embed a cryptographically strong message authentication code within the SYNCOOKIE data so as to be able to reject forged ACKCOOKIE suboptions.

Though an implementation MUST NOT send a SYNCOOKIE in any context except the SYN-ACK packet returned by a passive opener, implementations SHOULD accept SYNCOOKIEs in other contexts and reply with the appropriate ACKCOOKIE if possible.

4.4. Messages in the TCP datastream

If the use of tcpcrypt is negotiated between two hosts, they may then begin sending messages in the TCP datastream. These messages may or may not share boundaries with the TCP segments that transport them. We first provide a simple language for describing the data that must be exchanged, and then define the messages used for exchanging keys and for transmitting encrypted application data.

4.4.1. Datatypes and encodings

This document uses various datatypes to describe classes of values and the manner of encoding them for transmission in the TCP datastream.

4.4.1.1. Primitive and derived types

Primitive types include:

- "Byte", an octet. Its encoding is simply its value.
- "UInt16", an unsigned integer between 0 and 2^16 - 1, inclusive. Its encoding is two octets in network byte-order.
- "UInt64", an unsigned integer between 0 and 2^64 - 1, inclusive. Its encoding is eight octets in network byte-order.

Derived types include:

- A _tuple_ of two component types, written "type1, type2". Its encoding is the concatenation of the encodings of its components.
- A _vector_ containing multiple values of one element type. It may have arbitrary length, written "type[]", or a static length "n", written "type[n]". Its encoding is the concatenation of the encodings of its components.
- A _union_ of two alternative types, written "type1 | type2". Its encoding is the encoding of either of its types.
- An _encapsulation_, written "{ type }". Its encoding is the same as that of "UInt16, type", where the "UInt16" value gives the length of the encoding of the following "type" value.
- A _constant-bytes_ type, written as an all-capitalized word. This singleton type, a sub-type of "Byte[]", includes only the value assigned to its name in Appendix A.

4.4.1.2. Type definition

The notation "type-name ::= type-expr" defines the name "type-name" to represent the type expressed by "type-expr". Type names are always written in mixed case, with first letter capitalized.

4.4.1.3. Type declarations

The notation "type-expr value-name" declares "value-name" to be a value of type "type-expr". It is used to distinguish particular values for discussion. The declaration as a whole simply represents the type "type-expr".

4.4.1.4. Example

A datatype used in several places in this document is a vector of encapsulated messages; e.g.:

```
Messages ::= {Message}[]
```

If "Message" is a union type and thus may have variable length, the vector "Messages" may nevertheless be decoded completely without necessarily knowing how to decode every possible "Message" variant type, as the encapsulation prepends each message with its length.

This allows legacy implementations to operate insensitively to extensions which add variants to message types.

4.4.2. Frames

Each message sent in the TCP datastream is encapsulated in a "Frame":

```
Frame ::= { Init1
          | Init2
          | ApplicationFrame
          }
```

That is, a "Frame" is an encapsulation of an initialization message or of application data. These message types are defined below.

4.4.3. Key-exchange messages

Before application data may be sent, the INIT1 and INIT2 messages must be exchanged to negotiate session keys.

The key-exchange messages use constants to identify cryptographic algorithms. A "PubCipher" is a three-byte identifier for a public-key suite as specified in Figure 2:

```
    PubCipher ::= Byte[3]
```

A "SymCipher" is a four-byte symmetric algorithm specifier from Table 6:

```
    SymCipher ::= Byte[4]
```

The INIT1 message has the following type:

```
    Init1 ::= INIT1_MAGIC,
              PubCipher     pub-cipher,
              {SymCipher[]} sym-cipher-list,
              {Byte[]}      n_c,
              Byte[]        pk_c
```

The "pub-cipher" value is a selection among the entries of "pub-cipher-list" from the received PKCONF suboption, and determines both the length of "n_c" and the type of "pk_c".

The INIT2 message has the following type:

```
    Init2 ::= INIT2_MAGIC,
              SymCipher   sym-cipher,
              KeyMaterial kx_s
```

The "sym-cipher" value is a selection among the entries of "sym-cipher-list" from the received INIT1 message.

The type of the key material in "kx_s" depends on the public key cipher selected, as described in Section 3.5.

```
    KeyMaterial ::= KeyMaterialECDHE
                  | KeyMaterialOAEP
```

When ECDHE is used, the key material is encoded as follows:

```
    KeyMaterialECDHE ::= KEY_MATERIAL_ECDHE,
                         {Byte[]} n_s,
                         Byte[]   pk_s
```

The length of "n_s" depends on "pub-cipher" and is given in Figure 2. When OAEP+-RSA exp3 is used, the key material is simply a ciphertext in big-endian format:

```
    KeyMaterialOAEP ::= KEY_MATERIAL_OAEP,
                        Byte[] cipherText
```

4.4.4. Application frames

Once key-exchange succeeds and a host enters the ENCRYPTING phase, it may send frames of application data in the TCP datastream.

An "ApplicationFrame" comprises a header containing "public" control messages that are sent in the clear, together with a ciphertext portion containing the encryption of application data and private control messages:

```
ApplicationFrame ::= APPLICATION,
                     {Header}  header,
                     Byte[]    ciphertext
```

The header contains a vector of "PublicMessage" values:

```
Header ::= {PublicMessage}[]

PublicMessage ::= ApplicationOffset
                | ApplicationFin
                | Rekey
                | NonceClock
```

These public-message types are described below in Section 4.4.4.2, Section 4.4.4.4, and Section 4.4.4.1.

An application frame's ciphertext contains the encryption of a "PlainText" value:

```
PlainText ::= {PrivateMessage}[]

PrivateMessage ::= ApplicationData
                 | Urgent
                 | SyncReq
                 | SyncOk
```

These private-message types are described in Section 4.4.4.2 and Section 4.4.4.3.

4.4.4.1. Application frame security

The application frame's ciphertext value incorporates an "authentication tag" that protects the integrity of both the application data and the header; the decryption operation will fail if integrity has been compromised.

If an implementation receives a frame that fails to decrypt, it MUST ignore all public messages it did not need to process during the

decryption attempt. It MAY either terminate the connection, or ignore the frame and attempt further processing.

An "AssocData" value is related to an application frame and is authenticated as described in Section 3.6, but is not transmitted. It contains the frame's header:

```
AssocData ::= ASSOC_DATA, Header
```

The header may contain a "NonceClock" message, which provides a value that has never before been transmitted by the sending host in a distinct frame of this session:

```
NonceClock ::= FRAME_NONCE_CLOCK, UInt64 clock
```

A host MUST NOT ever transmit two distinct frames with the same "clock" value. A sending host SHOULD begin a session sending frames with the value set to zero, and increment the value at least once for each frame sent.

A "FrameNonce" value is related to an application frame and is used as an input to the encryption and decryption operations, but is not transmitted:

```
FrameNonce ::= FrameNonceClock | FrameNonceOffset
```

If a frame contains a "NonceClock" in its header, then "FrameNonce" takes this form:

```
FrameNonceClock ::= NonceClock, NONCE_PADDING
```

Above, the "NonceClock" value is the same as the one in the frame header, and "NONCE_PADDING" is three zero-valued bytes.

If a frame contains no "NonceClock" message, then the "FrameNonce" value instead takes this form:

```
FrameNonceOffset ::= ApplicationOffset, NONCE_PADDING
```

Above, the "ApplicationOffset" is the same as the one in the frame header, and "NONCE_PADDING" is three zero-valued bytes.

If a frame contains neither a "NonceClock" message nor an "ApplicationOffset" message, then the "FrameNonce" value is undefined and a receiving host MUST treat the frame the same as a frame whose decryption fails.

When constructing an outgoing application frame, a host MUST include a "NonceClock" message if the frame will contain no application data; that is, if it contains no "ApplicationData" message or contains an "ApplicationData" message with empty data. Furthermore, it MUST NOT ever transmit more than one frame containing application data for a particular "ApplicationOffset" value, unless the frames are identical.

The above requirements ensure that no two, distinct frames with the same "FrameNonce" value will ever be encrypted with the same key, in order to preserve the security properties of the authenticated encryption algorithm.

4.4.4.2. Application data messages

The following message transmits a portion of the application datastream.

```
ApplicationData ::= APPLICATION_DATA, Byte[] data
```

An implementation MUST NOT include more than one "ApplicationData" in any frame. For any frame which does contain one, it MUST also include an "ApplicationOffset" message, described below.

The following message provides an offset into the application datastream:

```
ApplicationOffset ::= APPLICATION_OFFSET, UInt64 offset
```

When it occurs in a frame containing an "ApplicationData", it indicates the location of that portion of data in the datastream. It may also occur in a frame with no application data. Its role in re-keying is described in Section 4.4.4.4.

The following message indicates that the application data contained in this frame lies at the end of the application datastream:

```
ApplicationFin ::= APPLICATION_FIN
```

An "ApplicationFin" message MUST be accompanied in the same frame by an "ApplicationOffset" message. These messages declare that the offset of the end of the datastream is the offset in the "ApplicationOffset" message plus the length of data in the accompanying "ApplicationData" message, or plus zero of there is none.

The URGENT message declares that the application data in this frame is "urgent", and that it belongs at the given offset into the (unframed) stream of application data.

```
Urgent ::= URGENT, UInt64 offset
```

4.4.4.3. Keep-alive and synchronization messages

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, Keep-Alive acknowledgments might not contain unique data. Hence, an old but cryptographically-valid acknowledgment could be replayed by an attacker to 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.)

The TCP Timestamps Option (TSopt) [RFC1323] could alternatively have been used to make Keep-Alives unique. However, because some middleboxes change the value of TSopt in packets, tcpcrypt does not protect the contents of the TCP TSopt option.

Instead, tcpcrypt uses the SYNC_REQ and SYNC_OK messages, protected by the enclosing frame's integrity protection, to probe connection liveness. This mechanism also provides a limited form of acknowledgement that is used in re-keying, as described in Section 4.4.4.4. These messages may be exchanged even when there is no application data to send.

The SYNC_REQ message has the following type:

```
SyncReq ::= SYNC_REQ, UInt64 clock
```

The "clock" value is a non-decreasing number. A host MUST increment "clock" at least once for every interval in which it sends a SYNC_REQ message. Implementations that support TSopt MAY choose to use the same value for "clock" that they would put in the TSval field of the TCP TSopt. However, implementations SHOULD "fuzz" any system clocks used to avoid disclosing either when a host was last rebooted or at what rate the hardware clock drifts.

A host that receives a SYNC_REQ message MUST reply with a SYNC_OK message, which has the following type:

```
SyncOk ::= SYNC_OK, UInt64 received-clock
```

The value of "received-clock" depends on the values of the "clock" fields in SYNC_REQ messages a host has received. A host must set "received-clock" to a value at least as high as the most recently received "clock", but no higher than the highest "clock" value received this session. If a host receives multiple frames with SYNC_REQ messages before it sends frames in the opposite direction, it SHOULD send a single SYNC_OK with "received-clock" set to the highest "clock" in the frames it has received.

4.4.4.4. Re-keying messages

During a tcpcrypt session, the set of keys used to perform authenticated encryption may be changed. This allows hosts to wipe from memory keys that could decrypt previously-transmitted frames. It also allows the use of message authentication codes that can safely protect only a limited number of messages.

The REKEY message is used while switching keys, and specifies which generation of keys has been used to encrypt and integrity-protect the current frame. The message has the following format:

```
Rekey ::= REKEY, Byte keyLSB
```

The byte "keyLSB" is the generation number of the keys used for the current frame, modulo 256.

Any frame containing a REKEY message MUST also contain an APPLICATION_OFFSET message, even if it contains no application data. The offset indicates the point at which data will be protected with the new key.

Once a host sends a REKEY message with a particular generation number and data offset, it MUST NOT use any previous generation of keys to encrypt frames carrying data at that offset or greater.

A host MAY use REKEY to increment the key generation number beyond the highest generation it knows the other side to be using. We call this process _initiating_ re-keying. When one host initiates re-keying, the other host MUST increment its key generation number to match, as described below (unless the other host has also simultaneously initiated re-keying).

A host MUST NOT initiate re-keying with any KeyLSB other than its current key number plus one modulo 256.

When initiating re-keying, a host MUST include, in the same frame as the REKEY message, a SYNC_REQ message with a fresh "clock" value; i.e., higher than any "clock" value it has previously transmitted.

We say that an outgoing frame is _acknowledged_ when the transmitter knows the remote side has received it: specifically, when the frame contained a SYNC_REQ with value "clock" and the transmitter later receives a SYNC_OK with a "received-clock" value at least as high as "clock".

When a host receives a frame containing a REKEY message, it MUST proceed as follows:

1. The receiver computes RECEIVE-KEY-NUMBER to be the closest integer to its own transmit key number that also equals "keyLSB" modulo 256. If no number is closest (because "keyLSB" is exactly 128 away from the transmit number modulo 256), the receiver MUST discard the frame. If RECEIVE-KEY-NUMBER is negative, the receiver MUST also discard the frame.

2. The receiver MUST authenticate and decrypt the frame using the receive keys with generation number RECEIVE-KEY-NUMBER. The receiver MUST discard the frame as usual if decryption fails.

3. If RECEIVE-KEY-NUMBER is greater than the receiver's current transmit key number, the receiver must wait to receive all frames with application data that precede the APPLICATION_OFFSET in the REKEY frame. Once it receives frames covering all this missing data (if any), it MUST increase its transmit number to RECEIVE-KEY-NUMBER and transmit a REKEY message. If the receiver has gotten multiple REKEY frames with different "keyLSB" values, it MUST increase its transmit key number to the highest RECEIVE-KEY-NUMBER of any frame for which it is not missing prior application data.

After sending a REKEY (whether initiating re-keying or just responding), a host MUST continue to send REKEY in all subsequent frames until one of those frames is acknowledged. This requirement allows tcpcrypt implementations to safely decrypt incoming frames out-of-order: any received frame that uses a new generation of keys will contain a REKEY message indicating the generation number, and frames with application data later in the stream can then be assumed to use this new generation.

A host SHOULD erase old transmit keys from memory once it has finished encrypting any outgoing frames with those keys, and old receive keys once it has decrypted all incoming application data prior to the offset of any REKEY message it has received.

A host MUST NOT initiate re-keying if it initiated a re-keying less than 60 seconds ago and has not transmitted at least 1 Megabyte (increased its application-data offset by 1,048,576) since the last re-keying. A host MUST NOT initiate re-keying if it has outstanding unacknowledged frames with REKEY messages for key numbers that are 127 or more below the current key. A host SHOULD NOT initiate more than one concurrent re-key operation if it has no data to send.

5. Examples

To illustrate the use of the CRYPT option in establishing a tcpcrypt session, consider the following ways in which a TCP connection may be established from host A to host B. We use notation S for SYN-only packet, SA for SYN-ACK packet, and A for packets with the ACK bit but not SYN bit. These examples are not normative.

5.1. Example 1: Normal handshake

```
(1) A -> B: S  CRYPT<>
(2) B -> A: SA CRYPT<PKCONF<0x200,0x201>>
(3) A -> B: A  data<Init1>
(4) B -> A: A  data<Init2>
(5) A -> B: A  data<ApplicationFrame,...>
```

(1) A indicates interest in using tcpcrypt. In (2), the server indicates willingness to use ECDHE with curves P256 and P521. Messages (3) and (4) complete the INIT1 and INIT2 key exchange messages described above, which are embedded in the data portion of the TCP segment. (5) From this point on, all messages are encrypted and integrity-protected inside application frames, which may or may not align with segment boundaries.

5.2. Example 2: Normal handshake with SYN cookie

```
(1) A -> B: S  CRYPT<>
(2) B -> A: SA CRYPT<PKCONF<0x200,0x201>, SYNCOOKIE<val>>
(3) A -> B: A  CRYPT<ACKCOOKIE<val>> data<Init1>
(4) B -> A: A  data<Init2,ApplicationFrame,...>
(5) A -> B: A  data<ApplicationFrame,...>
```

Same as previous example, except the server sends the client a SYN cookie value, which the client must echo in (3). Here also the application level protocol begins with B transmitting data, while in

the previous example, A was the first to transmit application-level data.

5.3. Example 3: tcpcrypt unsupported

```
(1) A -> B: S  CRYPT<>
(2) B -> A: SA
(3) A -> A: A
```

(1) A indicates interest in using tcpcrypt. (2) B does not support tcpcrypt, or a middle box strips out the CRYPT TCP option. (3) the client completes a normal three-way handshake, and tcpcrypt is not enabled for the connection.

5.4. Example 4: Reusing established state

```
(1) A -> B: S  CRYPT<NEXTK1<ID>>
(2) B -> A: SA CRYPT<NEXTK2>
(3) A -> B: A  data<ApplicationFrame,...>
```

(1) A indicates interest in using tcpcrypt with a session key derived from an existing key, to avoid the use of public key cryptography for the new session. (2) B supports tcpcrypt, has ID in its session ID cache, and is willing to proceed with session caching. (3) the client completes tcpcrypt's handshake within TCP's three-way handshake and tcpcrypt is enabled for the connection.

5.5. Example 5: Decline of state reuse

```
(1) A -> B: S  CRYPT<NEXTK1<ID>>
(2) B -> A: SA CRYPT<PKCONF<0x200,0x201>>
(3) A -> B: A  data<Init1>
(4) B -> A: A  data<Init2>
(5) A -> B: A  data<ApplicationFrame,...>
```

A wishes to use a key derived from a previous session key, but B does not recognize the session ID or has flushed it from its cache. Therefore, session establishment proceeds as in the first connection, using public key cryptography to negotiate a new series of session secrets (ss[i] values).

5.6. Example 6: Reversal of client and server roles

```
(1) A -> B: S  CRYPT<>
(2) B -> A: SA CRYPT<HELLO>
(3) A -> B: A  CRYPT<PKCONF<0x100>>
(4) B -> A: A  data<Init1>
(5) A -> B: A  data<Init2,ApplicationFrame,...>
```

Here the passive opener, B, wishes to play the role of the decryptor using RSA. By sending a HELLO suboption, B causes A to switch roles, so that now A is "S" and B plays the role of "C".

6. API extensions

The getsockopt call should have new options for IPPROTO_TCP:

TCP_CRYPT_SESSID -> returns the session ID and MUST return an error if tcpcrypt is in not in the ENCRYPTING state (e.g., because it has transitioned to DISABLED).

TCP_CRYPT_CMODE -> returns 1 if the local host played the "C" role in session key negotiation, 0 otherwise.

TCP_CRYPT_CONF -> returns the four-byte authenticated encryption algorithm in use by the connection (as specified in Table 6). In addition, implementations SHOULD provide the three-byte public key cipher (Figure 2) initially used to negotiate the session keys, as well as the public key length for algorithms with variable key sizes (e.g., OAEP+-RSA3).

TCP_CRYPT_PEER_SUPPORT -> returns 1 if the remote application is tcpcrypt-aware, as indicated by the remote host's use of a HELLO-app-support, HELLO-app-mandatory, or PKCONF-app-support CRYPT suboption (see Table 4).

TCP_CRYPT_FIN_RCVD -> returns 1 if an ApplicationFin message (see Section 4.4.4) has been received in this connection.

The setsockopt call should have:

TCP_CRYPT_CACHE_FLUSH -> setting this option to non-zero wipes cached session keys. Useful if application-level authentication discovers a man in the middle attack, to prevent the next connection from using NEXTK.

The following options should be readable and writable with getsockopt and setsockopt:

TCP_CRYPT_ENABLE -> one bit, enables or disables tcpcrypt extension on an unconnected (listening or new) socket.

TCP_CRYPT_CMODE_{DEFAULT,NEVER,ALWAYS}[_NK] -> As described in Section 4.2.

TCP_CRYPT_PKCONF -> set of allowed public key algorithms and CPRFs this host advertises in CRYPT PKCONF suboptions.

TCP_CRYPT_CCONF -> set of allowed symmetric ciphers and message authentication codes this host advertises in INIT1 messages.

TCP_CRYPT_SCONF -> order of preference of symmetric ciphers.

TCP_CRYPT_SUPPORT -> set to 1 if the application is tcpcrypt-aware. set to 2 if the application is tcpcrypt-aware and wishes to enter the DISABLED state if the remote application is not tcpcrypt-aware. An active opener SHOULD set the default value of 0 for each new connection. A passive opener SHOULD use a default value of 0 for each port, but SHOULD inherit the value of the listening socket for accepted connections. The behavior for each value is as follows:

When set to 0 The host MUST transition to the DISABLED state upon receiving a HELLO-app-mandatory option. The host MUST NOT send the HELLO-app-support, HELLO-app-mandatory, NEXTK2-app-support, or PKCONF-app-support options.

When set to 1 The "C" role host MUST use HELLO-app-support in place of the HELLO option, while the "S" role host MUST use the "PKCONF-app-support" in place of the "PKCONF" option. Either role must use NEXTK2-app-support in place of NEXTK2.

When set to 2 The "C" role host MUST use HELLO-app-mandatory option in place of the HELLO option, while the "S" role host MUST use "PKCONF-app-support" in place of the "PKCONF" option. Either role must use NEXTK2-app-support in place of NEXTK2. Either host MUST transition to DISABLED upon receipt of a HELLO or PKCONF option, but MUST proceed as usual in response to HELLO-app-support, HELLO-app-mandatory, and PKCONF-app-support.

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

- Default TCP_CRYPT_ENABLE value
- Default TCP_CRYPT_PKCONF value
- Default TCP_CRYPT_CCONF value
- Default TCP_CRYPT_SCONF value
- Types, key lengths, and regeneration intervals of local host's short-lived public keys

The session ID can be used for end-to-end security. For instance, applications might sign the session ID with public keys to

authenticate their ends of a connection. Because session IDs are not secret, servers can sign them in batches to amortize the cost of the signature over multiple connections. Alternatively, DSA signatures are cheaper to compute than to verify, so might be a good way for servers to authenticate themselves. A voice application could display the session ID on both parties' screens, and if they confirm by voice that they have the same ID, then the conversation is secure.

7. Acknowledgments

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

8. IANA Considerations

The following numbers need assignment by IANA:

o New TCP option kind number for CRYPT

A new registry entitled "tcpcrypt CRYPT suboptions" needs to be maintained by IANA as per the following table.

Symbol	Value
HELLO	0x01
HELLO-app-support	0x02
HELLO-app-mandatory	0x03
DECLINE	0x04
NEXTK2	0x05
NEXTK2-app-support	0x06
PKCONF	0x41
PKCONF-app-support	0x42
UNKNOWN	0x43
SYNCOOKIE	0x44
ACKCOOKIE	0x45
NEXTK1	0x84

TCP CRYPT suboptions.

Table 5

A "tcpcrypt Algorithm Identifiers" registry needs to be maintained by IANA as per the following table.

Algorithm Identifier	Value
Cipher: OAEP+-RSA with exponent 3	
min/max key size 2048/4096 bits ...	0x000100
min/max key size 4096/8192 bits ...	0x000101
min/max key size 8192/16384 bits ..	0x000102
min key size 16384 bits	0x000103
Extract: HKDF-Extract-SHA256	
CPRF: HKDF-Expand-SHA256	
N_C len: 32 bytes	
R_S len: 48 bytes	
K_LEN: 32 bytes	
Cipher: ECDHE-P256	0x000200
Extract: HKDF-Extract-SHA256	
CPRF: HKDF-Expand-SHA256	
N_C len: 32 bytes	
N_S len: 32 bytes	
K_LEN: 32 bytes	
Cipher: ECDHE-P521	0x000201
Extract: HKDF-Extract-SHA256	
CPRF: HKDF-Expand-SHA256	
N_C len: 32 bytes	
N_S len: 32 bytes	
K_LEN: 32 bytes	

TCP CRYPT algorithm identifiers.

Figure 2

A "tcpcrypt AEAD parameter" registry needs to be maintained by IANA as per the following table.

AEAD Algorithm	Key Length	sym-cipher
AEAD_AES_128_GCM	16 bytes	0x00000100
AEAD_AES_256_GCM	32 bytes	0x00000200

Authenticated-encryption algorithms corresponding to 4-byte sym-cipher specifiers in INIT1 and INIT2 messages. The use of encryption is described in Section 3.6. The algorithms are defined in [RFC5116].

Table 6

9. Security Considerations

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

If the application performs no authentication, then there are no guarantees against active attackers. Session IDs can be logged on both ends and man-in-the-middle attacks can be detected after the fact by comparing Session IDs offline.

Session IDs are not confidential.

tcpcrypt can be downgraded to regular TCP during the connection setup phase by removing any of the CRYPT options. The downgrade, and absence of protection, can of course be detected by the application as no Session ID will be returned.

tcpcrypt is not robust to the injection of FIN or RST packets. These will force the closure of the connection, but applications may probe the operating system to determine whether an authenticated end-of-stream has been signaled, thus avoiding semantic truncation attacks.

tcpcrypt uses short-lived keys to provide some forward secrecy. If a key is compromised all connections (new and cached) derived from that key will be compromised. The life of these keys should be kept to a minimum for stronger protection. A life of less than two minutes is recommended. Keys can be generated as frequently as practical, for example when servers have idle CPU time. For ECDHE-based key agreement, a new key can be chosen for each connection.

In the 4-way handshake, tcpcrypt does not have a key confirmation step. Hence, an active attacker can cause a connection to hang,

though this is possible even without tcpcrypt by altering sequence and ack numbers.

Attackers cannot force passive openers to move forward in their session caching chain without guessing the content of the NEXTK1 option, which will be hard without key knowledge.

10. References

10.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC1122] Braden, R., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, October 1989.

[RFC1323] Jacobson, V., Braden, B., and D. Borman, "TCP Extensions for High Performance", RFC 1323, May 1992.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2437] Kaliski, B. and J. Staddon, "PKCS #1: RSA Cryptography Specifications Version 2.0", RFC 2437, October 1998.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", January 2008.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, May 2010.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, January 2013.

10.2. Informative References

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

Appendix A. Protocol constant values

Value	Name
0x01	CONST_NEXTK
0x02	CONST_SESSID
0x03	CONST_REKEY
0x04	CONST_KEY_C
0x05	CONST_KEY_S
0x06	CONST_KEY_ENC
0x07	CONST_KEY_MAC
0x08	CONST_KEY_ACK
0x18	FRAME
0x20	FRAME_NONCE_CLOCK
0x21	FRAME_NONCE_OFFSET
0x000000	NONCE_PADDING
0x22	ASSOC_DATA
0x23	APPLICATION_DATA
0x24	APPLICATION_OFFSET
0x25	APPLICATION_FIN
0x26	URGENT
0x27	SYNC_REQ
0x28	SYNC_OK
0x29	REKEY
0x30	KEY_MATERIAL_ECDHE
0x31	KEY_MATERIAL_OAEP
0x15101a0e	INIT1_MAGIC
0x097105e0	INIT2_MAGIC
0x40	APPLICATION

Protocol constants.

Table 7

Authors' Addresses

Andrea Bittau
Stanford University
Department of Computer Science
353 Serra Mall, Room 288
Stanford, CA 94305
US

Phone: +1 650 723 8777
Email: bittau@cs.stanford.edu

Dan Boneh
Stanford University
Department of Computer Science
353 Serra Mall, Room 475
Stanford, CA 94305
US

Phone: +1 650 725 3897
Email: dabo@cs.stanford.edu

Daniel B. Giffin
Stanford University
Department of Computer Science
353 Serra Mall, Room 288
Stanford, CA 94305
US

Email: dbg@scs.stanford.edu

Mike Hamburg
Stanford University
Department of Computer Science
353 Serra Mall, Room 475
Stanford, CA 94305
US

Phone: +1 650 725 3897
Email: mike@shiftleft.org

Mark Handley
University College London
Department of Computer Science
University College London
Gower St.
London WC1E 6BT
UK

Phone: +44 20 7679 7296
Email: M.Handley@cs.ucl.ac.uk

David Mazieres
Stanford University
Department of Computer Science
353 Serra Mall, Room 290
Stanford, CA 94305
US

Phone: +1 415 490 9451
Email: dm@uun.org

Quinn Slack
Stanford University
Department of Computer Science
353 Serra Mall, Room 288
Stanford, CA 94305
US

Phone: +1 650 723 8777
Email: sqs@cs.stanford.edu