Internet Engineering Task Force   A. Bittau
Internet-Draft   D. Boneh
Intended status: Standards Track   M. Hamburg
Expires: March 1, 2012   Stanford University
                                                          M. Handley
                                           University College London
                                                         D. Mazieres
                                                           Q. Slack
                                                  Stanford University
                                                    August 29, 2011

              Cryptographic protection of TCP Streams (tcpcrypt)
                      draft-bittau-tcp-crypt-01.txt

Abstract

   This document presents tcpcrypt, a TCP extension for
   cryptographically protecting TCP segments. Tcpcrypt maintains the
   confidentiality of data transmitted in TCP segments against a passive
   eavesdropper. It can be used to protect already established TCP
   connections against denial-of-service attacks involving injection of
   forged RST segments or desynchronizing of sequence numbers. Finally,
   applications that perform authentication can obtain end-to-end
   confidentiality and integrity guarantees by tying authentication to
   tcpcrypt Session ID values.

   The extension defines two new TCP options, CRYPT and MAC, which are
   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. The MAC
   option carries a message authentication code with which hosts can
   verify the integrity of transmitted TCP segments. 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.

Status of this Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-
   Drafts is at http://datatracker.ietf.org/drafts/current/.

   Internet-Drafts are draft documents valid for a maximum of six months

and may be updated, replaced, or obsoleted by other documents at any
time. It is inappropriate to use Internet-Drafts as reference
material or to cite them other than as "work in progress."

This Internet-Draft will expire on March 1, 2012.

Copyright Notice

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:

o 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.

o Minimize computational cost, particularly on servers.

o Provide interfaces to higher-level software to facilitate end-to-end security, either in the application level protocol or after the fact. (E.g., client and server log session IDs and can compare them after the fact; if there was no tampering or eavesdropping, the IDs will match.)

o Be compatible with further extensions that allow authenticated resumption of TCP connections when either end changes IP address.

o Facilitate multipath TCP by identifying a TCP stream with a session ID independent of IP addresses and port numbers.

o 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 key exchange protocol 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

The setup phase negotiates use of the tcpcrypt extension. During
this phase, two hosts agree on a suite of cryptographic algorithms
and establish shared secret session keys.

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.
Importantly, 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. Applications MAY alternatively just
log Session IDs so as to enable attack detection after the fact
through comparison of the values logged at both ends.

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 if security
of the DISABLED state would not be adequate.

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 encrypt the Data portion of all

TCP segments transmitted and MUST include a Message Authentication
Code (MAC) in all segments transmitted. A host MUST furthermore
ignore any TCP segments received without the RST bit set, unless
those segments also contain a valid MAC.

A host MAY ignore RST segments without valid MACs. However,
operating systems SHOULD allow applications to control the dropping
of unMACed RST segments on a per-connection basis through an option
called TCP_CRYPT_RSTCHK option. Operating systems SHOULD furthermore
disable TCP_CRYPT_RSTCHK by default.

Once in the ENCRYPTING state, an endpoint MUST NOT directly or
indirectly transition to the DISABLED state under any circumstances.

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 two types of cryptographic algorithm:

o A _public key cipher_ is used with an ephemeral public key to
   exchange a random, shared secret. We use the notation
   ENC (K, VALUE) to denote an encryption of VALUE with public key K.

o A _collision-resistant pseudo-random function (CPRF)_ family is
   used to generate multiple cryptographic keys from a smaller shared
   secret. We use the notation CPRF (K, MESSAGE) to designate the
   output of the pseudo-random function identified by key K on
   MESSAGE.

Because public key ciphers and CPRFs often both make use of
cryptographic hashes, it generally makes sense to have both
algorithms based on the same hash function--for instance to pair the
OAEP+-RSA [RFC2437] cipher using a SHA-256-based mask-generation
function with the HMAC-SHA256 [RFC2104] CPRF. For this reason, the
public key cipher and CPRF are negotiated as a pair.

The encrypting phase employs two more types of algorithm:

o A _symmetric encryption algorithm_ is applied to all application
   data. The algorithm specifier includes both an underlying cipher
   (such as AES), and a mode of operation (such as CTR mode with TCP
   sequence numbers as the counter).

o A Message Authentication Code or _MAC_ is used to protect the
   contents of all TCP segments sent and received in the encrypting
   phase. The currently specified MACs handle data in a structured
   way so as to optimize authentication of TCP's Acknowledgment
   Number field in re-transmissions.

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 defeat 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

To establish shared session keys, tcpcrypt requires one host to
encrypt a secret value with the second host's public key. The second
host must subsequently use its private key to decrypt this value.
Thus, tcpcrypt's setup phase is asymmetric; the two hosts must play
different roles. We use "S" to denote the host that encrypts with
the other host's public key, and "C" to denote the host that decrypts
using its own private key.

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 costs 82 microseconds, while
decryption costs 10 milliseconds.

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, operating
systems MUST provide a mechanism for the passive opener to reverse
roles and play the "C" role, as discussed in Section 4.2.

3.4. Key exchange protocol

Every machine C has a short-lived public encryption key, K_C, which
gets refreshed periodically and SHOULD NOT ever be written to
persistent storage.

When a host C connects to S, the two engage in the following
protocol:

```
              C -> S:  HELLO
              S -> C:  PKCONF, pub-cipher-list
              C -> S:  INIT1, sym-cipher-list, N_C, K_C
              S -> C:  INIT2, sym-cipher, ENC (K_C, N_S)
```

Here the pub-cipher-list is a list of public key ciphers and key
lengths acceptable to the server. Sym-cipher-list specifies the
symmetric cipher suites acceptable to the client. N_C is a nonce
chosen at random by C, while K_C is C's public encryption key, which
MUST match one of the entries in pub-cipher-list. sym-cipher is the
symmetric cipher suite chosen by the server from sym-cipher-list.
Finally N_S is a "pre-session seed" chosen at random by S.

The two sides then compute a series of "session secrets" and
corresponding Session IDs as follows:

```
        param := { pub-cipher-list, sym-cipher-list, sym-cipher }

        ss[0] := CPRF (N_S, { K_C, param, N_C })
        ss[i] := CPRF (ss[i-1], TAG_NEXTK)

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

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 3.6.

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

$$mk[0] := CPRF (ss, TAG\_REKEY)$$
$$mk[i] := CPRF (mk[i-1], TAG\_REKEY)$$

Finally, each master key mk is used to generate four symmetric encryption keys:

```
kec := CPRF (mk, TAG_KEY_C_ENC || 1) || CPRF (mk, TAG_KEY_C_ENC || 2)
kac := CPRF (mk, TAG_KEY_C_MAC || 1) || CPRF (mk, TAG_KEY_C_MAC || 2)
kes := CPRF (mk, TAG_KEY_S_ENC || 1) || CPRF (mk, TAG_KEY_S_ENC || 2)
kas := CPRF (mk, TAG_KEY_S_MAC || 1) || CPRF (mk, TAG_KEY_S_MAC || 2)
```

The numbers 1 and 2 are each 1-byte byte long.

kec is used by the host in the "C" role to encrypt Data in transmitted TCP segments. If the symmetric encryption algorithm requires shorter keys, the key is truncated, keeping the left-most bytes only. Thus, if the symmetric cipher key length is less than or equal to the CPRF output length, a host need not compute CPRF (mk, TAG_KEY_C_ENC || 2).

kac is used by the host in the "C" role to compute MACs on transmitted segments, as described in Section 4.4. The key is truncated similarly to kec if the MAC requires a shorter key length. If the symmetric cipher is used in a mode that provides authentication as well as secrecy, kac need not be used.

kes and kas are used analogously to kec and kac for segments transmitted by the host in the "S" role.

3.5. Re-keying

We refer to the four encryption keys (kec, kac, kes, kas) 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 segments it transmits with a REKEY option containing j, so that the recipient host knows to check the MAC and decrypt the segment using the new keyset:

                    A -> B: REKEY<j>, MAC<...>, Data<...>

   Upon receiving a REKEY<j> segment, a recipient using transmit keys
   from a generation less than j must also update its transmit keys and
   start including a REKEY<j> option in all of its segments. A host
   must continue transmitting REKEY options until all segments with
   other generation numbers have been processed at both ends.

   Implementations MUST always transmit and retransmit identical
   ciphertext Data bytes for the same TCP sequence numbers. Thus, a
   retransmitted segment MUST always use the same keyset as the original
   segment. If the encryption algorithm requires an initialization
   vector, a retransmitted segment MUST additionally use the same
   initialization vector as the original segment. Hosts MUST NOT
   combine segments that were encrypted with different keysets or
   incompatible initialization vectors.

   Implementations SHOULD delete older-generation keys from memory once
   they have received all segments they will need to decrypt with the
   old keys and received acknowledgments for all segments they might
   need to retransmit.

3.6. 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 3.4 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 kec and kac
   for sending segments. Otherwise, if A had the "S" role originally,
   it uses kes and kas in the new session. 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 twelve

bytes to send NEXTK1 for the next connection. The behavior then
falls back to a normal public-key handshake.

3.6.1. 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 3.4. Applications SHOULD set
    TCP_CRYPT_CACHE_FLUSH whenever authentication of the session ID
    fails.

4. Extensions to TCP

The tcpcrypt extension adds two new kinds of option: CRYPT, and MAC.
Both are described in this section. During the setup phase, all TCP
segments MUST have the CRYPT option. In the ENCRYPTING state, all
segments MUST have the MAC option and may include the CRYPT option
for various purposes such as re-keying or keep-alive probes.

The idealized protocol of the previous section must be embedded in
the TCP handshake. Unfortunately, since the maximum TCP header size
is 60 bytes and the basic TCP header fields require 20 bytes, there
are at most 40 option payload bytes available, which is not enough to
hold the INIT1 and INIT2 messages. Tcpcrypt therefore uses the Data
portion of TCP segments to send the body of these messages.

Operating systems MUST keep track of which phase a data segment
belongs to, and MUST only deliver data to applications from segments
that are processed in the ENCRYPTING or DISABLED states.

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 1 shows which
TCP states an endpoint can be in for each tcpcrypt state.

```
+-------------+-------------------------+-------------------------+
| Tcpcrypt    | TCP states for an active | TCP states for a passive |
| state       | opener                  | 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 1

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. "rcv" and "snd" denote sending and receiving
segments, respectively. "any" means any possible event. "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. "mac" means
any valid TCP action, including no action, except that any segments
transmitted must be encrypted and contain a valid TCP MAC option. "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 "MAC" for a valid TCP MAC option, as
described in Section 4.4. 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. A segment MUST contain
the TCP MAC option if and only if Op is "MAC".

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.

```
            active OPEN              passive OPEN
            ------------ +----------+ ------------ +----------+
            snd S/NEXTK1 |  CLOSED  | x            |  LISTEN  |
    +------------------- |          |------------->|          |---------+
    |                    +----------+              +----------+         |
    |              +---+ |active OPEN              | |                  |
    |   rcv S/HELLO|   | |-----------   rcv S/HELLO| |    rcv S/NEXTK1|
    |   -----------|   | |snd S/HELLO ------------| |    -------------|
    V         drop| V V             snd SA/HELLO| |    snd SA/NEXTK2|
  +----------+    | +----------+                | |                  |
  |  NEXTK-  |___/ \|  HELLO-  |<------------------+ |                  |
  |  SENT    |     |  SENT    |                      |rcv S/HELLO      |
  +----------+     +----------+                      |-------------    |
   | | |             | |rcv S?/HELLO                 |snd SA/PKCONF    |
   | | |rcv S?/HELLO  | |-------------                      V          |
   | | |-------------  | |snd ?A/PKCONF    +----------+                |
   | | |snd ?A/PKCONF  | +--------------->|   S-MODE  |                |
   | | +---------------|----------------->|          |                |
   | +----------------+ |                  +----------+                |
   |      rcv SA/PKCONF| |rcv ?A/PKCONF          |                    |
   |      -------------| |-------------          |rcv A/INIT1          |
   |         snd A/INIT1| |snd A/INIT1           |-----------          |
   |                 V V                         |snd A/INIT2          |
   |               +----------+                  |                    |
   |rcv SA/NEXTK2  |  C-MODE  |       +---+ | +---+                    |
   |-------------  |          |  rcv */ |  | | |   |internal           |
   |snd A/MAC      +----------+  -------|  | | |   |or rcv */MAC        |
   | == or ==      |rcv A/INIT2   drop|  | | |   |or rcv R/           |
   |rcv S/NEXTK1   |-----------        |  V V V   |-----------         |
   |------------   |x                  +----------+ |mac               |
   |snd SA/NEXTK2  +------------------>|ENCRYPTING|-+                  |
   +------------------------------------>|          |<---------------+
                                        +----------+
```

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:

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

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

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

o 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.

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 the segment to contain a valid MAC, 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 the FIN. 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.

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.

A CLOSE event in the ENCRYPTING state MUST NOT change tcpcrypt's

state, only TCP's. The only valid tcpcrypt state transition from
ENCRYPTING is to CLOSED, which occurs only when TCP also transitions
to CLOSED.

[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 2. 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 2

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., use kec and kac for
   transmitted segments), 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 cause even the underlying TCP connection to fail.

. **The TCP CRYPT option**

   A CRYPT option has the following format:

```
            Byte    0       1       2         N
                  +-------+-------+-------..-------+
                  | Kind= |Length=|   Suboptions  |
                  |  OPT1 |   N   |               |
                  +-------+-------+-------..-------+

                   Format of TCP CRYPT option
```

   Kind is always OPT1. 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.

   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:

```
bit  0 1 2 3 4 5 6 7
    +-+-+-+-+-+-+-+-+
    |   Opcode =    |
    |0 0 x x x x x x|
    +-+-+-+-+-+-+-+-+
```

   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.

```
     0                   1
bit  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-------------------...
    |   Opcode =    |    Length =    |     N-2 bytes
    |0 1 x x x x x x|       N        |   of suboption data
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-------------------...
```

   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.

```
bit  0 1 2 3 4 5 6 7
     +-+-+-+-+-+-+-+-+-------...
     |    Opcode =   | data
     |1 0 x x x x x x|
     +-+-+-+-+-+-+-+-+-------...
```

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.

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. However, 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.

```
bit  0 1 2 3 4 5 6 7
     +-+-+-+-+-+-+-+-+-------...
     |    Opcode =   | data
     |1 1 x x x x x x|
     +-+-+-+-+-+-+-+-+-------...
```

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 3 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	INIT1	S
0x07	1	INIT2	S
0x41	*	PKCONF	S
0x42	*	PKCONF-app-support	S
0x43	*	UNKNOWN	SED
0x44	*	SYNCOOKIE	S
0x45	*	ACKCOOKIE	SED
0x80	5	SYNC_REQ	E
0x81	5	SYNC_OK	E
0x82	2	REKEY	E
0x83	6	REKEYSTREAM	E
0x84	10	NEXTK1	S
0x85	*	IV	E

Opcodes for suboptions of the TCP CRYPT option.

Table 3

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 OPT1, 3, 1 (Kind = OPT1, Length = 3, Suboption =
HELLO).

A host MUST enter the DISABLED state if, during the setup phase, it
receives a segment containing neither a TCP CRYPT nor a TCP MAC
option. 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. Once a host has entered

   DISABLED, it MUST NOT include the MAC option in any transmitted
   segment. 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.

[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:

   o 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.

   o 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.

   o 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: 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-ware.

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 3.6.

   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:

```
              Byte     0       1       2       3
                  +-------+-------+-------+-------+
                0 |Opcode |      Bytes 0-2        |
                  | 0x84  |       of SID[i]       |
                  +-------+-------+-------+-------+
                4 |            Bytes 3-6          |
                  |            of SID[i]          |
                  +-------+-------+-------+-------+
                8 |   Bytes 7-8   |
                  |   of SID[i]   |
                  +-------+-------+
```

                  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 use kec/kac for all sessions derived from the same pre-session seed, or kas/kes for all those sessions. 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 chose 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 the following format:

```
        Byte     0       1       2          N
               +-------+-------+-------..-------+
               |Opcode=|Length=|   Algorithm    |
               | 0x41  |   N   |   Specifiers   |
               +-------+-------+-------..-------+
```

                    Format of the PKCONF suboption

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

```
        0                   1                   2
 bit  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |0|  Algorithm  |  Min-Key-Len  |  Max-Key-Len  |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

        Format of algorithm specifier within PKCONF suboption

The following values of algorithm are defined:

```
 +-------+---------------------------+-------------+
 | Value | Cipher                    | CPRF        |
 +-------+---------------------------+-------------+
 |  0x01 | Rabin-Williams            | HMAC-SHA256 |
 |  0x02 | OAEP+-RSA with exponent 3 | HMAC-SHA256 |
 +-------+---------------------------+-------------+
```

(Values of algorithm over 127 are reserved for future use by multi-
byte algorithm specifiers for algorithms with fixed key sizes or more
compact min/max key length encodings.)

Hosts SHOULD implement Rabin-Williams, and MUST implement OAEP+-RSA3.
The Min-Key-Len and Max-Key-Len fields specify the minimum and
maximum key sizes acceptable for each particular algorithm. The
interpretation of the values of these fields depends on the
particular algorithm. For the two algorithms listed above, the two
Len values are expressed in terms of 256-bit (32-byte) key blocks, so
that, for example, the following algorithm specifier designates
Rabin-Williams keys with lengths from 1,024 to 8,192 bits.

```
        0                   1                   2
       0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |0 0 0 0 0 0 0 1|0 0 0 0 0 1 0 0|0 0 1 0 0 0 0 0|
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Depending on the value of the PKCONF suboption, it can either
indicate that the application is tcpcrypt-aware or not. This can be
used for bootstrapping application-level authentication without
requiring probing in upper layer protocols to check for support
(which may not be possible).

4.3.5. The UNKNOWN suboption

The UNKNOWN option has the following format:

```
        Byte    0      1      2                N
               +-------+-------+-------.......-------+
               |Opcode=|Length=| N-2 unknown one-byte |
               | 0x42  |   N   |   opcodes received   |
               +-------+-------+-------.......-------+

                  Format of the UNKNOWN suboption
```

This suboption is sent in response to an unknown 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 unknown 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 unknown option, it SHOULD reply with the
UNKNOWN suboption to notify the other side. If the host transitions
to DISABLED as a result of the unknown 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
unknown 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. If has the
following format:

```
            Byte     0      1      2         N
                  +-------+-------+-------..,-------+
                  |Opcode=|Length=|  N-2 bytes of   |
                  | 0x43  |   N   |   opaque data   |
                  +-------+-------+-------..,-------+
```

                    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. If has the
   following format:

```
            Byte     0      1      2              N
                  +-------+-------+-------..,-------+
                  |Opcode=|Length=|  N-2 bytes of   |
                  | 0x44  |   N   | SYNCOOKIE data  |
                  +-------+-------+-------..,-------+
```

                    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.

4.3.7. The SYNC_REQ and SYNC_OK suboptions

   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. Hence the SYNC_REQ and
   SYNC_OK suboptions allow the cryptographically protected TCP CRYPT
   option to contain unique data.

   The SYNC_REQ suboption is always 5 bytes, and has the following
   format:

```
            Byte    0       1       2       3       4
                 +-------+-------+-------+-------+-------+
                 |Opcode=|            Clock            |
                 | 0x80  |                             |
                 +-------+-------+-------+-------+-------+
```

                 Format of the SYNC_REQ suboption

   Clock is a 32-bit non-decreasing value. A host MUST increment Clock
   at least once for every interval in which it sends a Keep-Alive.
   Implementations that support TSopt MAY chose 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 suboption MUST reply with a SYNC_OK
   suboption, which is always five bytes and has the following format:

```
            Byte    0       1       2       3       4
                 +-------+-------+-------+-------+-------+
                 |Opcode=|        Received-Clock       |
                 | 0x81  |                             |
                 +-------+-------+-------+-------+-------+
```

                  Format of the SYNC_OK suboption

   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 delays acknowledgment of multiple packets with
   SYNC_REQ suboptions, it SHOULD send a single SYNC_OK with Received-

Clock set to the highest Clock in the packets it is acknowledging.

Because middleboxes sometimes "correct" inconsistent retransmissions, Keep-Alive segments with one byte of garbage data MUST use the same ciphertext byte as previously transmitted for that sequence number. Otherwise, a middlebox might change the byte back to its value in the original transmission, causing the cryptographic MAC to fail.

4.3.8. The REKEY and REKEYSTREAM suboptions

The REKEY and REKEYSTREAM suboptions are used to evolve encryption keys. Exactly one of the two options is valid with any given symmetric encryption algorithm and mode. Generally block ciphers will use REKEY while stream ciphers use REKEYSTREAM. We refer to a segment containing either option as a REKEY segment.

REKEY allows hosts to wipe from memory keys that could decrypt previously transmitted segments. It also allows the use of message authentication codes that are only secure up to a fixed number of messages. However, implementations MUST work in the presence of middleboxes that "correct" inconsistent data retransmissions. Hence, the value of ciphertext bytes must be the same in the original transmission and all retransmissions of a particular sequence number. This means a host MUST always use the same encryption key when transmitting or retransmitting the same range of sequence numbers. Re-keying only affects data transmitted in the future. Moreover, segments encrypted with different keysets MUST NOT be combined in retransmissions.

When switching keys, the REKEY suboption specifies which key set has been used to encrypt and integrity-protect the current segment. The suboption is always two bytes, and has the following format:

```
           Byte    0       1
                +-------+-------+
                |Opcode=|KeyLSB |
                | 0x83  |       |
                +-------+-------+
```

                Format of the REKEY suboption

KeyLSB is the generation number of the keys used to encrypt and MAC the current segment, modulo 256. REKEYSTREAM is the same as REKEY but includes the TCP Sequence Number offset at which the key change took effect, for cases in which decryption requires knowing how many bytes have been encrypted thus far with a key. To interoperate with middleboxes that rewrite sequence numbers, offsets from the Initial Sequence Number (ISN) are used instead of TCP sequence numbers

throughout tcpcrypt. The same occurs when dealing with
acknowledgement numbers.

```
        Byte     0       1       2       3       4       5
               +-------+-------+-------+-------+-------+-------+
               |Opcode=|KeyLSB |      Sequence Number Offset  |
               | 0x83  |       |            from ISN          |
               +-------+-------+-------+-------+-------+-------+
```

                  Format of the REKEYSTREAM suboption

A host MAY use REKEY to increment the session 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 blow (unless the other host has also
simultaneously initiated re-keying).

A host MAY initiate re-keying by including a REKEY suboption in a
syncable segment. A syncable segment is one that either contains
data, or is acknowledgment-only but contains a SYNC_REQ suboption
with a fresh Clock value--i.e., higher than any Clock value it has
previously transmitted. We say a syncable segment is _synced_ when
the transmitter knows the remote side has received it and all
previous sequence numbers. A data segment is synced when the
transmitter receives a cumulative acknowledgment for its sequence
number (a Selective Acknowledgment [RFC2018] is insufficient). An
acknowledgment-only segment is synced when the sender receives an
acknowledgment for its sequence number and a SYNC_OK with a high
enough Clock number.

A host MUST NOT initiate re-keying with an acknowledgment-only
segment that has either no SYNC_REQ suboption or a SYNC_REQ with an
old Clock value, because such a segment is not syncable. A host MUST
NOT initiate re-keying with any KeyLSB other than its current key
number plus one modulo 256.

When a host receives a segment containing a REKEY suboption, 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 segment. If RECEIVE-KEY-NUMBER is negative, the
    receiver MUST also discard the segment.

2. The receiver MUST authenticate and decrypt the segment using the
    receive keys with generation number RECEIVE-KEY-NUMBER. The
    receiver MUST discard the packet as usual if the MAC is invalid.

3. If RECEIVE-KEY-NUMBER is greater than the receiver's current
    transmit key number, the receiver must wait to receive all
    sequence numbers prior to the REKEY segment's. Once it receives
    segments covering all these missing sequence numbers (if any), it
    MUST increase its transmit number to RECEIVE-KEY-NUMBER and
    transmit a REKEY suboption. If the receiver has gotten multiple
    REKEY segments with different KeyLSB values, it MUST increase its
    transmit key number to the highest RECEIVE-KEY-NUMBER of any
    segment for which it is not missing prior sequence numbers.

After sending a REKEY (whether initiating re-keying or just
responding), a host MUST continue to send REKEY in all subsequent
segments until at least one of the following holds:

o One of the REKEY segments the host transmitted for its current
   transmit key number was syncable, and it has been synced.

o The host receives a cumulative acknowledgment for one of its REKEY
   segments with the current transmit key number, and the cumulative
   acknowledgment is in a segment encrypted with the new key but not
   containing a REKEY suboption.

A host SHOULD erase old keys from memory once the above requirements
are met.

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 sequence number by 1,048,576) since the last re-
keying. A host MUST NOT initiate re-keying if it has outstanding
unacknowledged REKEY segments 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.

4.3.9. The INIT1 and INIT2 suboptions

The INIT1 dataless suboption indicates that the Data portion of the
TCP segment contains the following data structure:

```
                 Byte    0       1       2       3
                   +-------+-------+-------+-------+
                   |    0x0001      | # sym ciphers |
                   +-------+-------+-------+-------+
                   |# bytes of N_C |# bytes of K_C |
                   +-------+-------+-------+-------+
                   |        symmetric cipher       |
                   :                               :
                   +-------+-------+-------+-------+
                   |               N_C             |
                   :                               :
                   +-------+-------+-------+-------+
                   |   0   |      type of K_C      |
                   +-------+-------+-------+-------+
                   |               K_C             |
                   :                               :
                   +-------+-------+-------+-------+
```

Each symmetric cipher has the following format:

```
    0                   1                   2                   3
bit  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |0 0 0 0 0 0 0 0| cipher alg    |0 0 0 0 0 0 0 0|    MAC alg    |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

         Format of symmetric cipher within INIT1 suboption

Note that the cipher and MAC are specified as a pair, using an 8-bit
identifier to specify each supported algorithm. The special cipher
value of 0 means that a MAC can be used with any cipher, or a cipher
with any MAC. The following cipher values are mandatory and MUST be
supported by all implementations:

```
          +-------+--------------------------------+
          | value | cipher                         |
          +-------+--------------------------------+
          |  0x00 | Any cipher okay with this MAC  |
          |  0x01 | AES-128 CTR mode w. seqno as IV |
          +-------+--------------------------------+
```

With AES-128 CTR mode, bytes are encrypted using their extended 64-
bit sequence number offset from the ISN. To encrypt the byte for
sequence number offset N:

o   Compute B = N - (N % 16).

o Let B* = 0^{128-|B|} || B be B with enough 0 bits pre-pended to
   make B* exactly 128 bits long.

o Let C = ENC-AES (ke[cs], B*).

o XOR the message byte with byte (N-B) of C.

The following MACs are mandatory and MUST be supported by all
implementations.

```
            +-------+-------------------------------+
            | value | MAC                           |
            +-------+-------------------------------+
            |  0x00 | Any MAC okay with this cipher |
            |  0x01 | HMAC-SHA2-128                 |
            +-------+-------------------------------+
```

The value "type of K_C" must be one of the public key specifiers
included earlier in the other host's PKCONF message.

The INIT2 dataless suboption indicates that the Data portion of the
TCP segment contains the following data structure:

```
            Byte    0       1       2       3
                +-------+-------+-------+-------+
                |    0x0002     |#byt ciphertext|
                +-------+-------+-------+-------+
                |    symmetric cipher suite     |
                +-------+-------+-------+-------+
                |            ciphertext         |
                :                               :
                +-------+-------+-------+-------+
```

Here the symmetric cipher suite is one selected by the host
transmitting the INIT2 segment, which will be playing the "S" role.
Neither the cipher nor the MAC may have value 0x00 in the INIT2
segment. The ciphertext is an encryption of N_S, as described in
Section 3.4.

Hosts MUST set the TCP PSH control bits on INIT1 and INIT2 segments.
Implementations MUST NOT set the TCP FIN control bit on INIT2
segments.

4.3.10. The IV suboption

The IV suboption is used to hold an initialization vector (IV) when
the negotiated encryption mode requires an initialization vector to
be transmitted with packets. It MUST NOT be included in transmitted

packets except in the ENCRYPTING state when the negotiated encryption
mode requires IVs. When the negotiated encryption mode does require
IVs, all segments transmitted in ENCRYPTING mode MUST contain an IV
suboption.

The IV suboption has the following format:

```
         Byte      0        1         N
               +-------+-------..-------+
               |Opcode=| Initialization |
               | 0x85  |     Vector     |
               +-------+-------..-------+
```

                   Format of the IV suboption

The length N of the IV is determined by the encryption algorithm and
mode negotiated.

As discussed in Section 4.3.8, a host MUST always transmit the same
ciphertext byte in retransmissions of a particular sequence number.
Thus, retransmitted segments must use the same IV each time.
Moreover, previously transmitted segments MUST NOT be combined on
retransmission if their IVs would prevent the ciphertext bytes from
remaining the same as in the original transmission.

4.4. The TCP MAC option

The MAC option is used to authenticate a TCP segment as described in
the next section. Once a host has entered the encrypting phase for a
session, the HOST must include a TCP MAC option in all segments it
sends. Furthermore, once in the encrypting phase, a host MUST ignore
any segments it receives that do not have a valid MAC option, except
for segments with the RST bit set if the application has not
requested cryptographic verification of RST segments.

The length of the MAC option is determined by the symmetric message
authentication code selected. The format of the MAC option is:

```
         Byte      0        1        2       N+1
               +-------+-------+------..,------+
               | Kind  | Len=  |    N-byte     |
               | OPT2  |  2+N  |      MAC      |
               +-------+-------+------..,------+
```

                   Format of TCP MAC option

The MAC is computed based on two data structures, a pseudo-packet
structure we call M, and an acknowledgment structure we call A. The

format of M is as follows:

```
                Byte    0       1       2       3
                    +-------+-------+-------+-------+
                 0 |    0x8000     |    length     |
                    +-------+-------+-------+-------+
                 4 |  off  | flags |    window     |
                    +-------+-------+---------------+
                 8 |    0x0000     |      urg      |
                    +-------+-------+-------+-------+
                12 |        seqno offset hi         |
                    +-------+-------+-------+-------+
                16 |         seqno offset           |
                    +-------+-------+-------+-------+
                20 |            options             |
                   :                                :
                    +-------+-------+-------+-------+
                   |      payload ciphertext        |
                   |     must be len-off bytes      |
                   :                                :
                    +-------+-------+-------+-------+
```

                        M data structure

The fields of M are defined as follows:

length
    Total size of the TCP segment from the start of the TCP header to
    the end of the IP datagram.

off
    Byte 12 of the TCP header (Data Offset)

flags
    Byte 13 of the TCP header (Control Bits)

window
    Bytes 14-15 of the TCP header (Window)

urg
    Bytes 18-19 of the TCP header (Urgent Pointer)

seqno offset hi
    Number of times the seqno offset field has wrapped from 0xffffff
    -> 0

seqno offset
      The low 32 bits of the sequence number offset (the Sequence Number
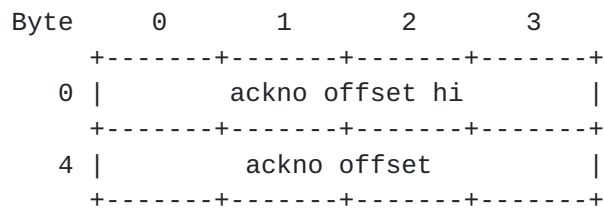   in the TCP header - ISN)

   options
      These are bytes 20-off of the TCP header. However, where the
      TSOPT (8), Skeeter (16), Bubba (17), MD5 (19), and MAC (OPT2)
      options appear, their contents (all but the kind and length bytes)
      are replaced with all zeroes.

   payload ciphertext
      This is the Data portion of the TCP segment, which contains
      encrypted ciphertext.

   The format of the A structure is as follows:

```
              Byte     0      1      2      3
                   +-------+-------+-------+-------+
                 0 |        ackno offset hi        |
                   +-------+-------+-------+-------+
                 4 |         ackno offset          |
                   +-------+-------+-------+-------+
```

                       A data structure

   The fields of A are defined as follows:

   ackno offset hi The number of times ackno offset hi has wrapped from
      0xffffff -> 0.

   ackno offset The lower 32 bits of the acknowledgement number offset
      from the remote end's ISN (TCP's acknowledgement header - ISN
      received).

   For HMAC-SHA2-128, The N-byte MAC value in the option contains the
   exclusive OR of MAC (M) and MAC (A).

5. Examples

   To illustrate these suboptions, consider the following series of 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<1, 4, 16>>
(3) A -> B: A  data<INIT1...>
(4) B -> A: A  data<INIT2...>
(5) A -> B: A  MAC<m> data<...>
```

   (1) A indicates interest in using tcpcrypt. In (2), the server
   indicates willingness to accept Rabin-Williams public keys between
   1,024 and 4,096 bytes long. 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 their integrity protected by a MAC option
   (described in the next section).

5.2. **Example 2: Normal handshake with SYN cookie**

```
(1) A -> B: S  CRYPT<>
(2) B -> A: SA CRYPT<PKCONF<1, 4, 16>, SYNCOOKIE<val>>
(3) A -> B: A  CRYPT<ACKCOOKIE<val>> data<INIT1...>
(4) B -> A: A  data<INIT2...>
(5) B -> A: A  MAC<m> data<...>
```

   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 by 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 -> A: A  MAC<m>
```

   (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, but does not does not have
ID in its session ID cache. (3) the client completes a normal three-
way handshake, and tcpcrypt is not 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<1, 4, 16>>
(3) A -> B: A  data<INIT1...>
(4) B -> A: A  data<INIT2...>
(5) A -> B: A  MAC<m> data<...>
```

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,
with public key encryption.

5.6. Exmaple 6: Reversal of client and server roles

```
(1) A -> B: S  CRYPT<>
(2) B -> A: SA CRYPT<HELLO>
(3) A -> B: A  CRYPT<PKCONF<1, 4, 16>>
(4) B -> A: A  data<INIT1...>
(5) A -> B: A  data<INIT2...>
(6) B -> A: A  MAC<m> data<...>
```

Here the server, B, wishes to play the role of the decryptor. By
sending a HELLO suboption, it causes A to switch roles, so that now A
is "S" and B can play the role of "C".

6. API extensions

The getsockopt call should have new options for IPPROTO_TCP:

   TCP_CRYPT_SESSID -> should return the session ID or error if no
   tcpcrypt.

   TCP_CRYPT_PUBKEY -> should return (mine, pubkey), where pubkey is
   the public key used to establish the session (K_C), and mine says
   whether the key belongs to this host or the remote peer.

   TCP_CRYPT_CONF -> returns encryption algorithms used for the
   current session.

   TCP_CRYPT_SUPPORT -> returns 1 if the remote application is
   tcpcrypt-aware.

The setsockopt call should have:

    TCP_CRYPT_CACHE_FLUSH -> setting 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_SECURST -> one bit, means ignore unauthenticated RST
    packets for this connection when set to 1.

    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 CRYPT INIT1 segments.

    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 requires the remote application
    to be tcpcrypt-aware.

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

o Default TCP_CRYPT_ENABLE value

o Default TCP_CRYPT_PKCONF value

o Default TCP_CRYPT_CCONF value

o Default TCP_CRYPT_SCONF value

o Types, key lengths, and regeneration intervals of local host's
   ephemeral 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. Alternative, 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.

Because the public key may change less often than once a session, it may alternatively be useful for the local end of a connection to authenticate itself by signing the local host's public key instead of the session ID.

7. Acknowledgments

This work was funded by gifts from Intel (to Brad Karp) and from Google, and by NSF award CNS-0716806 (A Clean-Slate Infrastructure for Information Flow Control).

8. IANA Considerations

When tcpcrypt is extended, the following numbers must be assigned by IANA:

o New opcodes for CRYPT suboptions

o New identifiers for public key algorithms

o New identifiers for symmetric key algorithms

This memo includes no request to IANA.

All drafts are required to have an IANA considerations section (see the update of RFC 2434 [I-D.narten-iana-considerations-rfc2434bis] for a guide). If the draft does not require IANA to do anything, the section contains an explicit statement that this is the case (as above). If there are no requirements for IANA, the section will be removed during conversion into an RFC by the RFC Editor.

9. Security Considerations

All drafts are required to have a security considerations section. See RFC 3552 [RFC3552] for a guide.

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.

   [RFC2018] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP
              Selective Acknowledgment Options", RFC 2018, October 1996.

   [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.

10.2. Informative References

   [I-D.narten-iana-considerations-rfc2434bis]
              Narten, T. and H. Alvestrand, "Guidelines for Writing an
              IANA Considerations Section in RFCs",
              draft-narten-iana-considerations-rfc2434bis-09 (work in
              progress), March 2008.

   [RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC
              Text on Security Considerations", BCP 72, RFC 3552,
              July 2003.

Appendix A. Protocol constant values

```
              +-------+---------------+
              | Value | Name          |
              +-------+---------------+
              |  0x01 | TAG_NEXTK     |
              |  0x02 | TAG_SESSID    |
              |  0x03 | TAG_REKEY     |
              |  0x04 | TAG_KEY_C_ENC |
              |  0x05 | TAG_KEY_C_MAC |
              |  0x06 | TAG_KEY_S_ENC |
              |  0x07 | TAG_KEY_S_MAC |
              +-------+---------------+
```

                     Protocol constants.

                          Table 4

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

   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