Cryptographic protection of TCP Streams (tcpcrypt)
draft-bittau-tcpinc-tcpcrypt-00.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 protects connections against denial-of-service
   attacks involving desynchronizing of sequence numbers, and when
   enabled, against forged RST segments. 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

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

This Internet-Draft will expire on April 29, 2015.

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 [RFC6824] 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.
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 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 option.

A host SHOULD accept RST segments without valid MACs by default. However, the application SHOULD be allowed to force unMACed RST segments to be dropped by enabling the TCP_CRYPT_RSTCHK option on the connection.

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 three types of cryptographic algorithms:

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

o 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.

o A _collision-resistant pseudo-random function (CPRF)_ is used to
   generate multiple cryptographic keys from a pseudo-random key,
   typically the output of the extract function. We use the notation
   CPRF (K, TAG, L) to designate the output of L bytes of the pseudo-
   random function identified by key K on TAG. A collision-resistant
   function is one on which, for sufficiently large L, an attacker
   cannot find two distinct inputs K_1, TAG_1 and K_2, TAG_2 such
   that CPRF (K_1, TAG_1, L) = CPRF (K_2, TAG_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, TAG, L) -> OKM
           T(0) = empty string (zero length)
           T(1) = HMAC-Hash(PRK, T(0) | TAG | 0x01)
           T(2) = HMAC-Hash(PRK, T(1) | TAG | 0x02)
           T(3) = HMAC-Hash(PRK, T(2) | TAG | 0x03)
           ...

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

The symbol | denotes concatenation, and the counter concatenated with
TAG 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 all application data. This mode authenticates both
application data and most of the TCP header (excepting header fields
commonly modified by middleboxes).

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

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, 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 or key agreement parameter, PK_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, pub-cipher, PK_C
        S -> C:  INIT2, sym-cipher, KX_S
```

The parameters are defined as follows:

o pub-cipher-list: a list of public key ciphers and parameters acceptable to S. These are defined in Figure 3.

o sym-cipher-list: a list of symmetric cipher suites acceptable to
   C. These are specified in Table 6 as parameters for ASM, discussed
   in Section 3.6.

o N_C: Nonce chosen at random by C.

o pub-cipher: the type of PK_C.

o PK_C: C's public key or key agreement parameter.

o sym-cipher: the symmetric cipher selected by S.

o 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 3. 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 complete data payload from the TCP segments
with the INIT1 and INIT2 suboptions (detailed in Section 4.3.9).

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 TAG values are constants defined in Table 7. The
K_LEN values and nonce sizes are negotiated, and are specified in
Figure 3.

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

```
        mk[0] := CPRF (ss, CONST_REKEY | 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_len)
        k_sc := CPRF (mk, CONST_KEY_S, ae_len)
```

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
since in tcpcrypt's threat model, a connection to an adversary can be
made and so keys need not be verified. If an erroneous key
negotiation that yields two different keys occurs, all subsequent
packets will be dropped due to an incorrect MAC, causing the TCP
connection to hang. This is not a threat because in plain TCP, an
active attacker could have modified sequence and ack numbers to hang
the connection anyway.

3.5. Data encryption and authentication

tcpcrypt encrypts and authenticates all application data. It also
authenticates some parts of the TCP header. There are several TCP-
specific constraints with regards to authenticated encryption that
tcpcrypt must meet for performance and compatibility with
middleboxes:

o The ciphertext for a particular byte position in tcpcrypt's
   sequence must never change, even if reencryption occurs after
   coalescing and retransmission. This is because a middlebox may
   discard a changed payload on retransmission.

o Authentication must occur only on fields not modified by
   middleboxes. In particular, port numbers must not be
   authenticated, and sequence and ack numbers must be authenticated
   according to an offset from the initial sequence number, because
   these can be modulated by a middlebox.

o An efficient mechanism is needed for recomputing the
   authentication tag when only the ack numbers change. For example,
   on retransmissions, the authenticated encryption authentication
   tag can be efficiently updated without having to recompute the tag
   on the entire packet payload.

Authenticated encryption modes such as GCM do not meet these
criteria. For example, even with identical plaintext, ciphertext
values depend on the byte position at which one starts encrypting a
segment. Hence two small segments will appear to have different
content from their coalesced counterpart; middleboxes might drop such
coalesced retransmissions after falsely detecting subterfuge attacks.
Furthermore, existing authenticated encryption modes do not allow
efficient updating of the authentication tag when only small parts of
the data have changed. A new mode is needed to meet all these
constraints, and we introduce _Authenticated Sequence Mode_ (ASM) in
Section 3.6 as a solution.

ASM takes three parameters: a cipher, a MAC and an ACK MAC. At a
high-level, the cipher is used to encrypt the TCP payload in counter

mode, using a counter derived from TCP's sequence number. The MAC
covers the ciphertext and parts of the TCP header. The ACK MAC
covers the ACK numbers and is XORed with the previously computed MAC
to produce the authenticated encryption authentication tag. This tag
can be quickly updated if only the ACK numbers have changed. This
approach is principled because ACK messages are conceptually separate
from data packets, so MACing them separately is appropriate. In TCP,
ACKs are piggybacked to data segments merely as an optimization.

XORing two PRF-based MACs together was shown secure by Katz and
Lindell [aggregate-macs].

3.6. Authenticated Sequence Mode (ASM)

ASM is parameterized by a cipher, MAC and ACK MAC. The operations
supported by ASM are:

ASM-Encrypt (PRK, Seq, Message, Assoc-Data, Up-Data) ->
            (Ciphertext, Auth-Tag)

ASM-Decrypt (PRK, Seq, Cipher-Text, Assoc-Data, Up-Data, Auth-Tag) ->
            { (Valid, Message) OR
              (Invalid,)
            }

ASM-Update (PRK, Up-Data-Prev, Up-Data-New, Auth-Tag-Prev) ->
            Auth-Tag

The arguments and return values are:

o _PRK_ a pseudo-random key.

o _Seq_ the byte position in the stream of Message or Cipher-Text.
   In tcpcrypt, this is an extended version of TCP's sequence number.

o _Message_ the Message to encrypt. In tcpcrypt, this is TCP's
   payload.

o _Assoc-Data_ the associated data to be MACed but not encrypted.
   In tcpcrypt, this contains parts of the TCP header.

o _Up-Data_ the updatable data to be MACed but not encrypted, that
   can also be efficiently updated and reMACed. In tcpcrypt, this
   will cover an extended version of TCP's ACK numbers.

o _Ciphertext_ the encrypted version of Message.

o _Auth-Tag_ the authenticated encryption authentication tag. In
   tcpcrypt, this will be the MAC option.

ASM-Decrypt returns one of the constants Valid or Invalid, depending
on whether the authentication tag can be verified successfully or
not. For Valid inputs, the Message is returned as well.

The PRK supplied to ASM is expanded into keys used for individual
operation as follows:

            k_enc := CPRF (PRK, CONST_KEY_ENC, cipher-key-len)
            k_mac := CPRF (PRK, CONST_KEY_MAC, mac-key-len)
            k_ack := CPRF (PRK, CONST_KEY_ACK, ack-mac-key-len)

The next sections describe ASM operations in detail.

3.6.1. ASM-Encrypt

The interface to encrypt is as follows:

        ASM-Encrypt (PRK, Seq, Message, Assoc-Data, Up-Data) ->
                        (Ciphertext, Auth-Tag)

Keys (denoted by k_*) are derived from PRK as explained in
Section 3.6.

The following steps occur:

1. Message is encrypted to produce Ciphertext using the cipher in
    counter mode. Seq is the counter and k_enc is the key. When
    encrypting Seq, its value must always be a multiple of the
    cipher's block size. In the event that the message does not
    begin on an even block boundary, Seq must be rounded down,
    encrypted, and leading bytes of its encryption discarded.

2. The MAC is run over the concatenation of Ciphertext and Assoc-
    Data to produce MAC1, using k_mac as the key.

3. The ACK MAC is run over Up-Data to produce MAC2, using k_ack as
    the key.

4. MAC1 and MAC2 are XORed to produce Auth-Tag.

Using AES-128 as an example, encryption in counter mode using Seq as
the counter happens as follows.

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

o Let B* = 0^{128-|B|} | B be B in network (big-endian) byte order
   with enough 0 bits pre-pended to make B* exactly 128 bits long.

o Let C = ENC-AES (k_enc, B*).

o Discard the first (Seq-B) bytes on C and begin byte-by-byte XORing
   the remaining portion with the message.

o Continue computing ENC-AES (k_enc, B* + 16), ENC-AES (k_enc, B* +
   32), etc. to generate enough bytes to XOR with the whole message.

If AES-128 is used as the ACK MAC, the Ack number (64-bit extended,
offset from ISN) is first padded on the left with enough zeros to
produce a 128-bit big-endian value. The number is then encrypted
using AES.

3.6.2. ASM-Decrypt

The interface to decrypt is as follows:

```
ASM-Decrypt (PRK, Seq, Cipher-Text, Assoc-Data, Up-Data, Auth-Tag) ->
             { (Valid, Message)  OR
               (Invalid, )
```

Keys (denoted by k_*) are derived from PRK as explained in
Section 3.6.

The following steps occur:

1. The MAC is run over the concatenation of Ciphertext and Assoc-
    Data to produce MAC1, using k_mac as the key.

2. The ACK MAC is run over Up-Data to produce MAC2, using k_ack as
    the key.

3. MAC1 and MAC2 are XORed and compared to Auth-Tag. If different,
    the process stops and the constant Invalid is returned along with
    no message. Otherwise the process continues.

4. Ciphertext is decrypted to produce Message using the cipher in
    counter mode. Seq is the counter and k_enc is the key. The
    Valid constant is returned along with Message.

3.6.3. ASM-Update

   The interface to update the authenticated encryption authentication
   tag is as follows:

       ASM-Update (PRK, Up-Data-Prev, Up-Data-New, Auth-Tag-Prev) ->
                       Auth-Tag

   Keys (denoted by k_*) are derived from PRK as explained in
   Section 3.6.

   The following steps occur:

   1. The ACK MAC is run over Up-Data-Prev using k_ack to produce MAC2-
       Prev.

   2. MAC2-Prev is XORed with Auth-Tag-Prev to produce MAC1.

   3. The ACK MAC is run over Up-Data to produce MAC2, using k_ack as
       the key.

   4. MAC1 and MAC2 are XORed to produce Auth-Tag.

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 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. Hosts MUST NOT combine segments that were encrypted with
different keysets.

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 that would
need to be retransmitted encrypted under 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 3.4 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 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 3.4. 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 is 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 (after the SYN exchanges) 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 2 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 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. "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.

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.

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:

```
              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.
   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:

```
   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|
    +-+-+-+-+-+-+-+-+-------...
```
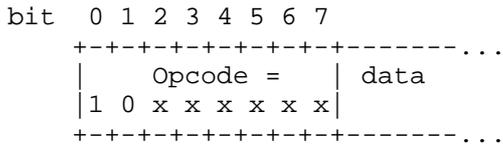
If a host sees an unknown opcode in this range, it MUST ignore the
suboption and all subsequent suboptions in the same TCP CRYPT option.
However, if more than one CRYPT option appears in the TCP header, the
host MUST continue processing suboptions from the next TCP CRYPT
option. Skipping suboptions in 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.

```
bit  0 1 2 3 4 5 6 7
    +-+-+-+-+-+-+-+-+-------...
    |     Opcode =  | data
    |1 1 x x x x x x|
    +-+-+-+-+-+-+-+-+-------...
```
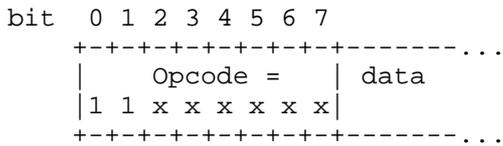
Should a host encounter an unknown opcode greater than or equal to
0xc0 during the setup phase of the protocol, the host MUST transition
to the DISABLED state. It SHOULD respond with both a DECLINE
suboption and an UNKNOWN suboption specifying the opcode of the
unknown mandatory suboption, after which 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 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      |
|  0x07 |      1 | INIT1              | S      |
|  0x08 |      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      |
+-------+--------+--------------------+--------+
```

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 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. 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
reply without a TCP CRYPT option to disable tcpcrypt. 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.

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:

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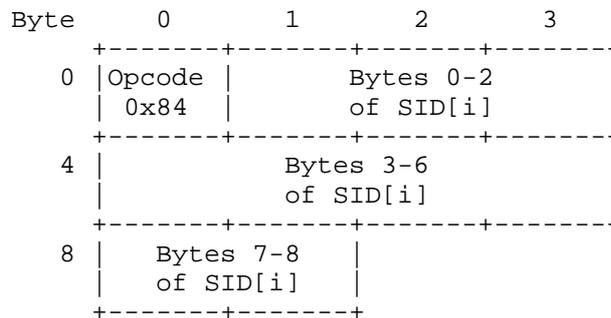
A passive opener SHOULD send a DECLINE suboption in response to a HELLO suboption or NEXTK1 suboption in a received SYN segment if it supports tcpcrypt but does not wish to engage in encryption for this particular session.

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:

```
            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 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
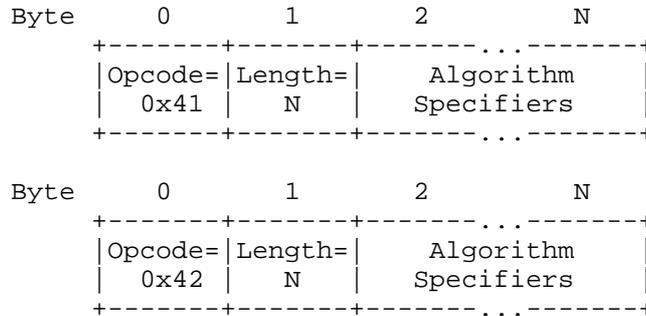   segment containing a NEXTK1 suboption.

   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 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

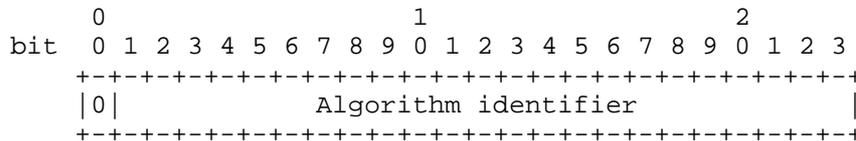connection, as described in Section 4.2.1.

4.3.4. The PKCONF suboption

   The PKCONF option has one of the following two formats:

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

            Byte    0       1       2           N
                 +-------+-------+-------...-------+
                 |Opcode=|Length=|   Algorithm    |
                 | 0x42  |   N   |   Specifiers   |
                 +-------+-------+-------...-------+
```

                 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:

```
             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 identifier              |
           +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

   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 3.

   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:

```
     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. It 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. It 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 for any given
symmetric encryption algorithm. All algorithms in Table 6 use the
REKEY option. REKEYSTREAM is reserved for future use should tcpcrypt
evolve to support a stream cipher. 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 |
                       | 0x82  |       |
                       +-------+-------+
```

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
acknowledgment numbers.

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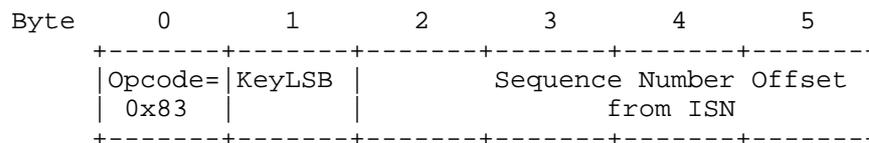
                  Format of the REKEYSTREAM suboption

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 below (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
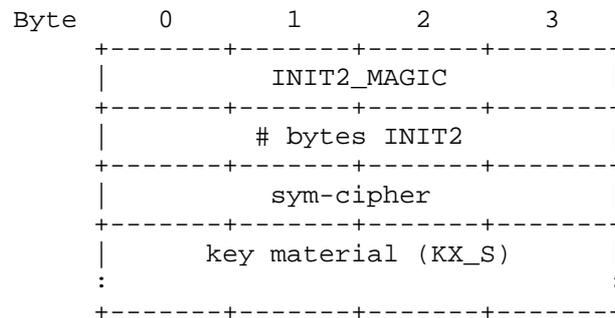concurrent re-key operation if it has no data to send.

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
                   +-------+-------+-------+-------+
                   |           INIT1_MAGIC         |
                   +-------+-------+-------+-------+
                   |          # bytes INIT1        |
                   +-------+-------+-------+-------+
                   |   0   |       pub-cipher      |
                   +-------+-------+-------+-------+
                   |       0       | # sym-ciphers |
                   +-------+-------+-------+-------+
                   |        sym-cipher-list        |
                   :                               :
                   +-------+-------+-------+-------+
                   |              N_C              |
                   :                               :
                   +-------+-------+-------+-------+
                   |              PK_C             |
                   :                               :
                   +-------+-------+-------+-------+
```
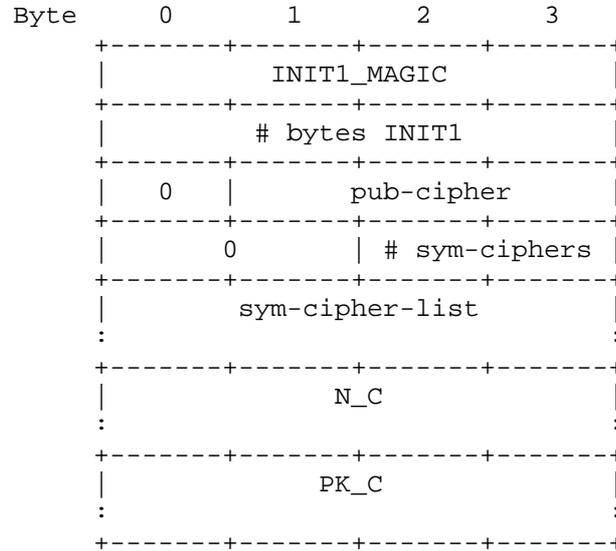
   The constant INIT1_MAGIC is specified in Table 7. # bytes INIT1
   specifies the length of the entire INIT1 structure, including the
   four-byte INIT1_MAGIC that precedes the length. pub-cipher is a
   three-byte public key suite as specified in Figure 3, which specifies
   both the length of N_C and the type of PK_C. sym-cipher-list is a
   list of four-byte symmetric algorithm specifiers from Table 6. Of
   those listed, 0x00000100 (AES-128 / HMAC-SHA-256-128 / AES-128) is
   MANDATORY to implement, and the others OPTIONAL. # sym-ciphers
   specifies the number of four-byte entries in this list.

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

```
Byte    0       1       2       3
     +-------+-------+-------+-------+
     |          INIT2_MAGIC          |
     +-------+-------+-------+-------+
     |         # bytes INIT2         |
     +-------+-------+-------+-------+
     |          sym-cipher           |
     +-------+-------+-------+-------+
     |       key material (KX_S)     |
     :                               :
     +-------+-------+-------+-------+
```

Format of the INIT2 suboption

Figure 2

The INIT2_MAGIC constant is specified in Table 7. # bytes INIT2 is
the total length of the INIT2 structure, including the 4-byte
INIT2_MAGIC constant preceding the length. sym-cipher specifies which
entry of sym-cipher-list from the INIT1 message the host transmitting
the INIT2 segment has selected.

The key material depends on the public key cipher selected, as
described in Section 3.4. When ECDHE is used, key material is
encoded as follows:

```
    +---...---+---...---+
    |   N_S   |   PK_S  |
    |         |         |
    +---...---+---...---+
```
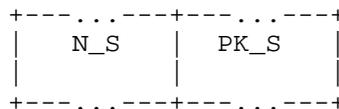
The length of N_S depends on pub-cipher and is given in Figure 3.
PK_S uses the rest of the message. When OAEP+-RSA exp3 is used, KX_S
is simply a ciphertext in big-endian format.

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

4.4. The TCP MAC option

The MAC option is used to authenticate a TCP segment. 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 the authentication tag as output from authenticated
encryption. Apart from payload, two headers are included in the
authenticated encryption process: a pseudo-header structure we call
Assoc-Data, and an acknowledgment structure we call Up-Data. The
format of Assoc-Data 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            |
        :                               :
        +-------+-------+-------+-------+
```

                Assoc-Data data structure

The fields of Assoc-Data 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 0xffffffff
    -> 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), TCP-AO (29), and
    MAC (OPT2) options appear, their contents (all but the kind and
    length bytes) are replaced with all zeroes.

The format of the Up-Data structure is as follows:

```
            Byte     0       1       2       3
                  +-------+-------+-------+-------+
               0  |       ackno offset hi        |
                  +-------+-------+-------+-------+
               4  |        ackno offset          |
                  +-------+-------+-------+-------+
```
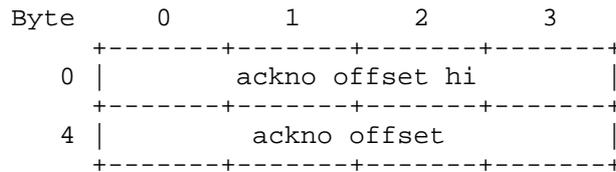
                     Up-Data data structure

The fields of Up-Data are defined as follows:

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

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

The two structures, Assoc-Data and Up-Data, are used in ASM to
calculate the TCP MAC option. All multi-byte values are encoded in
big-endian format.

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<0x200,0x201>>
(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 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 their integrity protected by a MAC option.

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...>
(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, 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  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,
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...>
(6) B -> A: A  MAC<m> data<...>
```

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 3) 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).

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_RSTCHK -> 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 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:

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
   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

8. IANA Considerations

   The following numbers need assignment by IANA:

   o New TCP option kind number for CRYPT

   o New TCP option kind number for MAC

   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 |
                 | INIT1 | 0x07 |
                 | INIT2 | 0x08 |
                 | PKCONF | 0x41 |
                 | PKCONF-app-support | 0x42 |
                 | UNKNOWN | 0x43 |
                 | SYNCOOKIE | 0x44 |
                 | ACKCOOKIE | 0x45 |
                 | SYNC_REQ | 0x80 |
                 | SYNC_OK | 0x81 |
                 | REKEY | 0x82 |
                 | REKEYSTREAM | 0x83 |
                 | NEXTK1 | 0x84 |
                 | IV | 0x85 |
                 +--------------------+-------+

                       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 3

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

```
+---------+------------------+---------+------------+
| Cipher  | MAC              | ACK MAC | Sym-cipher |
+---------+------------------+---------+------------+
| AES-128 | HMAC-SHA-256-128 | AES-128 | 0x00000100 |
| AES-128 | Poly1305-AES-128 | AES-128 | 0x00000200 |
| AES-128 | CMAC-AES-128     | AES-128 | 0x00000300 |
+---------+------------------+---------+------------+
```

ASM parameters corresponding to 4-byte sym-cipher specifiers in INIT1
and INIT2 messages. ASM itself is specified in Section 3.6. HMAC-
SHA-256-128 is HMAC-SHA-256 with a 128-bit key and output truncated
to 128 bits.

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.

   By default tcpcrypt does not protect against RST packet injection.
   The connection must be configured with TCP_CRYPT_RSTCHK enabled to
   protect against malicious (unMACed) RSTs.

   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.

   [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.

   [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

   [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.

   [aggregate-macs]
              Katz, J. and A. Lindell, "Aggregate Message Authentication
              Codes", Topics in Cryptology - CT-RSA , 2008.

   [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
0x15101a0e	INIT1_MAGIC
0x097105e0	INIT2_MAGIC

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

   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

                       Inner Space for TCP Options
                    draft-briscoe-tcpm-inner-space-01

Abstract

   This document describes an experimental method to extend the limited
   space for control options in every segment of a TCP connection. It
   can use a dual handshake so that, from the very first SYN segment,
   extra option space can immediately start to be used optimistically.
   At the same time a dual handshake prevents a legacy server from
   getting confused and sending the control options to the application
   as user-data. The dual handshake is only one strategy - a single
   handshake will usually suffice once deployment has got started. The
   protocol is designed to traverse most known middleboxes including
   connection splitters, because it sits wholly within the TCP Data. It
   also provides reliable ordered delivery for control options.
   Therefore, it should allow new TCP options to be introduced i) with
   minimal middlebox traversal problems; ii) with incremental deployment
   from legacy servers; iii) without an extra round of handshaking delay
   iv) without having to provide its own loss recovery and ordering
   mechanism and v) without arbitrary limits on available space.

Copyright Notice

Table of Contents

1. Introduction

   TCP has become hard to extend, partly because the option space was
   limited to 40B when TCP was first defined [RFC0793] and partly
   because many middleboxes only forward TCP headers that conform to the
   stereotype they expect.

   This specification ensures new TCP capabilities can traverse most
   middleboxes by tunnelling TCP options within the TCP Data as 'Inner
   Options' (Figure 1). Then the TCP receiver can reconstruct the Inner
   Options sent by the sender, even if the middlebox resegments the data
   stream and even if it strips 'Outer' options from the TCP header that
   it does not recognise. The two words 'Inner Space' are appropriate
   as a name for the scheme; 'Inner' because it encapsulates options
   within the TCP Data and 'Space' because the space within the TCP Data
   is virtually unlimited--constrained only by the maximum segment size.

```
 ,-----.                   TCP Payload                ,-----.
 | App |<-------------------------------------------->| App |
 |-----|                                              |-----|
 |     |       Inner Options within TCP Data          |     |
 |     |<-------------------------------------------->|     |
 |     |                                              |     |
 | TCP | TCP Header and            TCP header and     | TCP |
 |     | Outer Options  ,---------. Outer Options     |     |
 |     |<-------------->|Middlebox|<--------------->|     |
 |-----|                |---------|                   |-----|
 | IP  |                |  IP     |                   | IP  |
 :     :                :         :         :         :     :
```
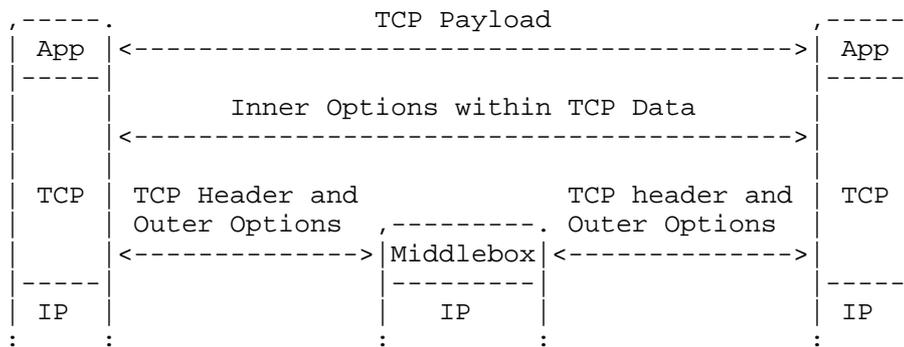
                    Figure 1: Encapsulation Approach

   TCP options fall into three main categories:

   a. Those that have to remain as Outer Options--typically those
       concerned with transmission of each TCP segment, e.g. Timestamps
       and Selective ACKnowledgements (SACK);

   b. Those that are best as Inner Options--typically those concerned
       with transmission of the data as a stream, e.g. the TCP
       Authentication Option [RFC5925] or tcpcrypt [I-D.bittau-tcpinc];

   c. Those that can be either Inner or Outer Options--typically those
       used at the start of a connection which is also inherently the
       start of the first segment so segmentation is not a concern.

   Pressure of space is most acute in the initial segments of each half-
   connection, i.e. the SYN and SYN/ACK, and particularly the SYN. Even
   though Inner Space is not suitable for category (a) options, moving
   all of categories (b) and (c) into Inner Space frees up plenty of
   outer space in the header for category (a).

   The following list of options that might be required on a SYN
   illustrates how acute the problem is:

   o 4B: Maximum Segment Size (MSS) [RFC0793];

   o 2B: SACK-ok [RFC2018];

   o 3B: Window Scale [RFC7323];

   o 10B: Timestamp [RFC7323];

   o 12B: Multipath TCP [RFC6824];

   o 6-18B: TCP Fast Open on a resumed connection
      [I-D.ietf-tcpm-fastopen];

   o 16B: TCP-AO [RFC5925];

   There is probably potential for compressing together multiple options
   in order to mitigate the option space problem. However, the option
   space problem has to be faced, because complex special placement is
   already being contemplated for options that can be larger than 40B on
   their own (e.g. the key agreement options of tcpcrypt
   [I-D.bittau-tcpinc]).

   Given the Inner Space protocol places control options within TCP
   Data, it is critical that a legacy TCP receiver is never confused
   into passing this mix to an application as if it were pure data.
   Naively, both ends could handshake to check they understand the
   protocol, but this would introduce a round of delay and it would not
   solve the shortage of space in a SYN. Instead, the client uses dual
   handshakes; one suitable for an upgraded server, and the other for an
   ordinary server. Then, if the client discovers that the server does
   not understand the new protocol, it can abort the upgraded handshake
   before the server passes corrupt data to the application. Otherwise,
   if the server does understand the new protocol, the client can abort
   the ordinary handshake. Either way, it has added zero extra delay.
   Interworking of the dual handshake with TCP Fast Open
   [I-D.ietf-tcpm-fastopen] is carefully defined so that either server
   can pass data to the application as soon as the initial SYN arrives.

   When control options are placed within the TCP Data they inherently
   get delivered reliably and in order. Although this was not
   originally recognised as part of the design brief, it offers the
   significant benefit of simplifying the design of new TCP options.
   Reliable ordered delivery no longer has to be individually crafted
   into the design of each new TCP option.

   Solving the five problems of i) option-space exhaustion; ii)
   middlebox traversal; iii) legacy server confusion; iv) reliable
   ordered control message delivery; and v) handshake latency; does not
   come without cost:

   o So that the Inner Space protocol is immune to option stripping, it
      flags its presence using a magic number within the TCP Data of the
      initial segment in each direction, not a conventional TCP option
      in the header. This introduces a risk that payload in an ordinary
      SYN or SYN/ACK might be mistaken for the Inner Space protocol (an
      initial worst-case estimate of the probability is one connection
      globally every 40 years). Nonetheless, the risk is zero in the
      (currently common) case of an ordinary connection without payload

during the handshake. There is also no risk of a mistake the
other way round--an upgraded connection cannot be mistaken for an
ordinary connection.

o Although the dual handshake introduces no extra latency, it
   introduces extra connection processing & state, extra traffic and
   extra header processing. Initial estimates put the percentage
   overhead in single digits for connection processing and state, and
   traffic overhead at only a few hundredths of a percent.
   Nonetheless, once the most popular TCP servers have upgraded, only
   a single handshake will be necessary most of the time and overhead
   should drop to vanishingly small proportions.

Finally, it should be noted that the ambition of this work is more
than just an incrementally deployable, low latency way to extend TCP
option space. The aim is to move towards a more structured way for
middleboxes to interact transparently with, rather than arbitrarily
interfere with, end-system TCP stacks. This has been achieved for
connection and stream control options, but it will still be hard to
introduce new per-segment control options, which will still have to
be located within the traditional Outer TCP Options.

1.1. Motivation for Adoption Now (to be removed before publication)

It seems inevitable that ultimately more option space will be needed,
particularly given that many of the TCP options introduced recently
consume large numbers of bits in order to provide sufficient
information entropy, which is not amenable to compression.

Extension of TCP option space requires support from both ends. This
means it will take many years before the facility is functional for
most pairs of end-points. Therefore, given the problem is already
becoming pressing, a solution needs to start being deployed now.

1.2. Scope

This experimental specification extends the TCP wire protocol. It is
independent of the dynamic rate control behaviour of TCP and it is
independent of (and thus compatible with) any protocol that
encapsulates TCP, including IPv4 and IPv6.

1.3. Experiment Goals

TCP is critical to the robust functioning of the Internet, therefore
any proposed modifications to TCP need to be thoroughly tested.

Success criteria: The experimental protocol will be considered
   successful if it satisfies the following requirements in the

consensus opinion of the IETF tcpm working group. The protocol
needs to be sufficiently well specified so that more than one
implementation can be built in order to test its function,
robustness, overhead and interoperability (with itself, with
previous version of TCP, and with various commonly deployed
middleboxes). Non-functional issues such as recommendations on
message timing also need to be tested. Various optional
extensions to the protocol are proposed in Appendix A so
experiments are also needed to determine whether these extensions
ought to remain optional, or perhaps be removed or become
mandatory.

Duration: To be credible, the experiment will need to last at least
12 months from publication of the present specification. If
successful, it would then be appropriate to progress to a
standards track specification, complemented by a report on the
experiments.

1.4. Document Roadmap

The body of the document starts with a full specification of the
Inner Space extension to TCP (Section 2). It is rather terse,
answering 'What?' and 'How?' questions, but deferring 'Why?' to
Section 3. The careful design choices made are not necessarily
apparent from a superficial read of the specification, so the Design
Rationale section is fairly extensive. The body of the document ends
with Section 4 that checks possible interactions between the new
scheme and pre-existing variants of TCP, including interaction with
partial implementations of TCP in known middleboxes.

Appendix A specifies optional extensions to the protocol that will
need to be implemented experimentally to determine whether they are
useful. And Appendix B discusses the merits of the chosen design
against alternative schemes.

1.5. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
"SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
document are to be interpreted as described in [RFC2119]. In this
document, these words will appear with that interpretation only when
in ALL CAPS. Lower case uses of these words are not to be
interpreted as carrying RFC-2119 significance.

TCP Header: As defined in [RFC0793]. Even though the present
specification places TCP options beyond the Data Offset, the term
'TCP Header' is still used to mean only those fields at the head
of the segment, delimited by the TCP Data Offset.

Inner TCP Options (or just Inner Options): TCP options placed in the
   space that the present specification makes available beyond the
   Data Offset.

Outer TCP Options (or just Outer Options): The TCP options in the
   traditional location directly after the base TCP Header and before
   the TCP Data Offset.

Prefix TCP Options: Inner Options to be processed before the Outer
   Options.

Suffix TCP Options: Inner Options to be processed after the Outer
   Options.

TCP options: Any TCP options, whether inner, outer or both. This
   specification makes this term on its own ambiguous so it should be
   qualified if it is intended to mean TCP options in a certain
   location.

TCP Payload: Data to be passed to the layer above TCP. The present
   specification redefines the TCP Payload so that it does not
   include the Inner TCP Options, the Inner Space Option and any
   Magic Number, even though they are located beyond the Data Offset.

TCP Data: The information in a TCP segment after the Data Offset,
   including the TCP Payload, Inner TCP Options, the Inner Space
   Option and the Magic Number defined in the present specification.

client: The process taking the role of actively opening a TCP
   connection.

server: The process taking the role of TCP listener.

Upgraded Segment: A segment that will only be fully understood by a
   host complying with the present specification (even though it
   might appear valid to a pre-existing TCP receiver). Similarly,
   Upgraded SYN, Upgraded SYN/ACK etc.

Ordinary Segment: A segment complying with pre-existing TCP
   specifications but not the present specification. Similarly,
   Ordinary SYN, Ordinary SYN/ACK etc.

Upgraded Connection: A connection starting with an Upgraded SYN.

Ordinary Connection: A connection starting with an Ordinary SYN.

   Upgraded Host: A host complying with the present document as well as
      with pre-existing TCP specifications. Similarly Upgraded TCP
      Client, Upgraded TCP Server, etc.

   Legacy Host: A host complying with pre-existing TCP specifications,
      but not with the present document. Similarly Legacy TCP Client,
      Legacy TCP Server, etc.

   Note that the term 'Ordinary' is used for segments and connections,
   but the term 'Legacy' is used for hosts. This is because, if the
   Inner Space protocol were widely used in future, a host that could
   not open an Upgraded Connection would be considered deficient and
   therefore 'Legacy', whereas an Ordinary Connection would not be
   considered deficient in the future; because it will always be
   legitimate to open an Ordinary Connection if extra option space is
   not needed.

2. Protocol Specification

2.1. Protocol Interaction Model

2.1.1. Dual 3-Way Handshake

   During initial deployment, an Upgraded TCP Client sends two
   alternative SYNs: an Ordinary SYN in case the server is legacy and a
   SYN-U in case the server is upgraded. The two SYNs MUST have the
   same network addresses and the same destination port, but different
   source ports. Once the client establishes which type of server has
   responded, it continues the connection appropriate to that server
   type and aborts the other without completing the 3-way handshake.

   The format of the SYN-U will be described later (Section 2.2.2). At
   this stage it is only necessary to know that the client can put
   either TCP options or payload (or both) in a SYN-U, in the space
   traditionally intended only for payload. So if the server's response
   shows that it does not recognise the Upgraded SYN-U, the client is
   responsible for aborting the Upgraded Connection. This ensures that
   a Legacy TCP Server will never erroneously confuse the application by
   passing it TCP options as if they were user-data.

   Section 3.1 explains various strategies the client can use to send
   the SYN-U first and defer or avoid sending the Ordinary SYN.
   However, such strategies are local optimizations that do not need to
   be standardized. The rules below cover the most aggressive case, in
   which the client sends the SYN-U then the Ordinary SYN back-to-back
   to avoid any extra delay. Nonetheless, the rules are just as
   applicable if the client defers or avoids sending the Ordinary SYN.

Table 1 summarises the TCP 3-way handshake exchange for each of the
two SYNs in the two right-hand columns, between an Upgraded TCP
Client (the active opener) and either:

1. a Legacy Server, in the top half of the table (steps 2-4), or

2. an Upgraded Server, in the bottom half of the table (steps 2-4)

Because the two SYNs come from different source ports, the server
will treat them as separate connections, probably using separate
threads (assuming a threaded server). A load balancer might forward
each SYN to separate replicas of the same logical server. Each
replica will deal with each incoming SYN independently - it does not
need to co-ordinate with the other replica.

```
+------+----------------+-------------------+-------------------+
|      |                | Ordinary          | Upgraded          |
|      |                | Connection        | Connection        |
+------+----------------+-------------------+-------------------+
| 1    | Upgraded Client | >SYN             | >SYN-U            |
|      |                 |                   |                   |
| /\/\ | /\/\/\/\/\/\/\/\ | /\/\/\/\/\/\/\/\ | /\/\/\/\/\/\/\/\  |
| 2    |  Legacy Server  | <SYN/ACK         | <SYN/ACK          |
|      |                 |                   |                   |
| 3a   | Upgraded Client | Waits for response|                   |
|      |                 | to both SYNs      |                   |
|      |                 |                   |                   |
| 3b   |       "         | >ACK             | >RST              |
|      |                 |                   |                   |
| 4    |                 | Cont...          |                   |
|      |                 |                   |                   |
| /\/\ | /\/\/\/\/\/\/\/\ | /\/\/\/\/\/\/\/\ | /\/\/\/\/\/\/\/\  |
| 2    | Upgraded Server | <SYN/ACK         | <SYN/ACK-U        |
|      |                 |                   |                   |
| 3a   | Upgraded Client | Waits for response|                   |
|      |                 | to SYN-U          |                   |
|      |                 |                   |                   |
| 3b   |       "         | >RST             | >ACK              |
|      |                 |                   |                   |
| 4    |                 |                   | Cont...           |
+------+----------------+-------------------+-------------------+
```

            Table 1: Dual 3-Way Handshake in Two Server Scenarios

Each column of the table shows the required 3-way handshake exchange
within each connection, using the following symbols:

   > means client to server;

    < means server to client;

    Cont... means the TCP connection continues.

The connection that starts with an Ordinary SYN is called the
'Ordinary Connection' and the one that starts with a SYN-U is called
the 'Upgraded Connection'. An Upgraded Server MUST respond to a
SYN-U with an Upgraded SYN/ACK (termed a SYN/ACK-U and defined in
Section 2.2.2). Then the client recognises that it is talking to an
Upgraded Server. The client's behaviour depends on which response it
receives first, as follows:

o If the client first receives a SYN/ACK response on the Ordinary
   Connection, it MUST wait for the response on the Upgraded
   Connection. It then proceeds as follows:

   * If the response on the Upgraded Connection is an Ordinary SYN/
      ACK, the client MUST reset (RST) the Upgraded Connection and it
      can continue with the Ordinary Connection.

   * If the response on the Upgraded Connection is an Upgraded SYN/
      ACK-U, the client MUST reset (RST) the Ordinary Connection and
      it can continue with the Upgraded Connection.

o If the client first receives an Ordinary SYN/ACK response on the
   Upgraded Connection, it MUST reset (RST) the Upgraded Connection
   immediately. It can then wait for the response on the Ordinary
   Connection and, once it arrives, continue as normal.

o If the client first receives an Upgraded SYN/ACK-U response on the
   Upgraded Connection, it MUST reset (RST) the Ordinary Connection
   immediately and continue with the Upgraded Connection.

2.1.2. Dual Handshake Retransmission Behaviour

If the client receives a response to the SYN, but a short while after
that {ToDo: duration TBA} the response to the SYN-U has not arrived,
it SHOULD retransmit the SYN-U. If latency is more important than
the extra TCP option space, in parallel to any retransmission, or
instead of any retransmission, the client MAY give up on the Upgraded
(SYN-U) Connection by sending a reset (RST) and completing the 3-way
handshake of the Ordinary Connection.

If the client receives no response at all to either the SYN or the
SYN-U, it SHOULD solely retransmit one or the other, not both. If
latency is more important than the extra TCP option space, it will
retransmit the SYN. Otherwise it will retransmit the SYN-U. It MUST

NOT retransmit both segments, because the lack of response could be
due to severe congestion.

2.1.3. Continuing the Upgraded Connection

Once an Upgraded Connection has been successfully negotiated in the
SYN, SYN/ACK exchange, either host can allocate any amount of the TCP
Data space in any subsequent segment for extra TCP options. In fact,
the sender has to use the upgraded segment structure in every
subsequent segment of the connection that contains non-zero TCP
Payload. The sender can use the upgraded structure in a segment
carrying no user-data (e.g. a pure ACK), but it does not have to.

As well as extra option space, the facility offers other advantages,
such as reliable ordered delivery of Inner TCP Options on empty
segments and more robust middlebox traversal. If none of these
features is needed, at any point the facility can be disabled for the
rest of the connection, using the ModeSwitch TCP option in
Appendix A.1. Interestingly, the ModeSwitch options itself can be
very simple because it uses the reliable ordered delivery property of
Inner Options, rather than having to cater for the possibility that a
message to switch to disabled mode might be lost or reordered.

2.2. Upgraded Segment Structure and Format

2.2.1. Structure of an Upgraded Segment

An Upgraded Segment is structured as shown in Figure 2. Up to the
TCP Data Offset, the structure is identical to an Ordinary TCP
Segment, with a base TCP Header (BaseHdr) and the usual facility to
set the Data Offset (DO) to allow space for TCP options. These
regular TCP options are renamed by this specification to Outer TCP
Options or just Outer Options, and labelled as OuterOpts in the
figure.

The first segment in each direction (i.e. the SYN or the SYN/ACK) is
identifiable as upgraded by the presence of the 4-octet Magic Number
A (MagicA) at the start of the TCP Data. The probability that an
Upgraded Server will mistake arbitrary data at the beginning of the
payload of an Ordinary Segment for the Magic Number has to be allowed
for, but it is vanishingly small (see Section 3.2.1). Once an
Upgraded Connection has been negotiated during the SYN - SYN/ACK
exchange, a magic number is not needed to identify Upgraded Segments,
because both ends know that the protocol requires the sender to use
the upgraded format on all subsequent segments with non-zero TCP
Data. Aside from the magic number, the structure of the rest of an
Upgraded Segment is effectively the same whether a) SYN=1 or b)
SYN=0.

```
                                             |    SOO    |
  a) SYN=1                                   ,--------->|
  |          DO          |  1  |   Len  |                InOO     |   SPS   |
  ,------------------>,------>,------->,-------------------->,------->|
  +--------+----------+-------+--------+----------+----------+--------+
  | BaseHdr| OuterOpts| MagicA| InSpace|PrefixOpts|SuffixOpts| Payload|
  +--------+----------+-------+--------+----------+----------+--------+
                      |                 '----------.----------'       |
                      |                     Inner Options             |
                      '----------------------.----------------------'
                                        TCP Data


  b) SYN=0
  |          DO          | Len |              InOO      |   SPS   |
  ,------------------>,------->,-------------------->,------->|
  +--------+----------+--------+----------------------+--------+
  | BaseHdr| OuterOpts| InSpace|    Inner Options     | Payload|
  +--------+----------+--------+----------------------+--------+
                      '---------------.----------------------'
                               TCP Data
```
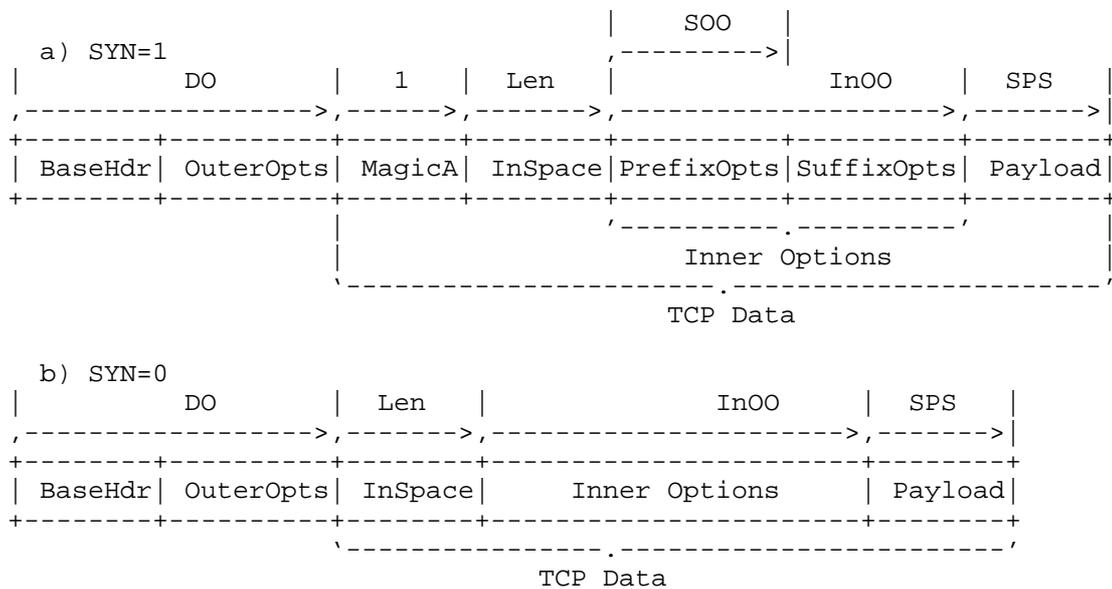
       All offsets are specified in 4-octet (32-bit) words, except SPS,
                          which is in octets.

        Figure 2: The Structure of an Upgraded Segment (not to scale)

Unlike an Ordinary TCP Segment, the Payload of an Upgraded Segment
does not start straight after the TCP Data Offset. Instead, Figure 2
shows that space is provided for additional Inner TCP Options before
the TCP Payload. The size of this space is termed the Inner Options
Offset (InOO). The TCP receiver reads the InOO field from the Inner
Option Space (InSpace) option defined in Section 2.2.2.

The InSpace Option is located in a standardized location so that the
receiver can find it:

o On a segment with SYN=1, an Upgraded TCP Sender MUST locate the
   InSpace Option straight after the magic number, specifically 4 *
   (DO + 1) octets from the start of the segment.

o On a segment with SYN=0, an Upgraded TCP Sender MUST locate the
   InSpace Option at the beginning of the TCP Data, specifically 4 *
   DO octets from the start of the segment.

Because the InSpace Option is only ever located in a standardized
location it does not need to follow the RFC 793 format of a TCP
option. Therefore, although we call InSpace an 'option', we do not
describe it as a 'TCP option'.

The Sent Payload Size (SPS) is also read from within the InSpace
Option. If the byte-stream has been resegmented, it allows the
receiver to step from one InSpace Option to the next even if the
InSpace Options are no longer at the start of each segment (see
Section 2.3).

On a segment with SYN=1 (i.e. a SYN or SYN/ACK) the Suffix Options
Offset (SOO) is also read from within the InSpace Option. It
delineates the end of the Prefix TCP Options (PrefixOpts in the
figure) and the start of the Suffix TCP Options (SuffixOpts). When
SYN=1, the receiver processes PrefixOpts before OuterOpts, then
SuffixOpts afterwards. When SYN=0, the receiver processes the Outer
Options before the Inner Options. Full details of option processing
are given in Section 2.3.

2.2.2. Format of the InSpace Option

The internal structure of the InSpace Option for an Upgraded SYN or
SYN/ACK segment (SYN=1) is defined in Figure 3a) and for a segment
with SYN=0 in Figure 3b).

```
 0                   1                   2                   3
 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
  a) SYN = 1
 +----------------------------+------------------------+---+
 |    Sent Payload Size (SPS) |Inner Options Offset (InOO)|Len|
 +----------------------------+------------------------+---+
 |        Magic Number B      |Suffix Options Offset (SOO)|CU |
 +----------------------------+------------------------+---+

  b) SYN = 0
 +----------------------------+------------------------+---+
 |    Sent Payload Size (SPS) |Inner Options Offset (InOO)|Len|
 +----------------------------+------------------------+---+
```
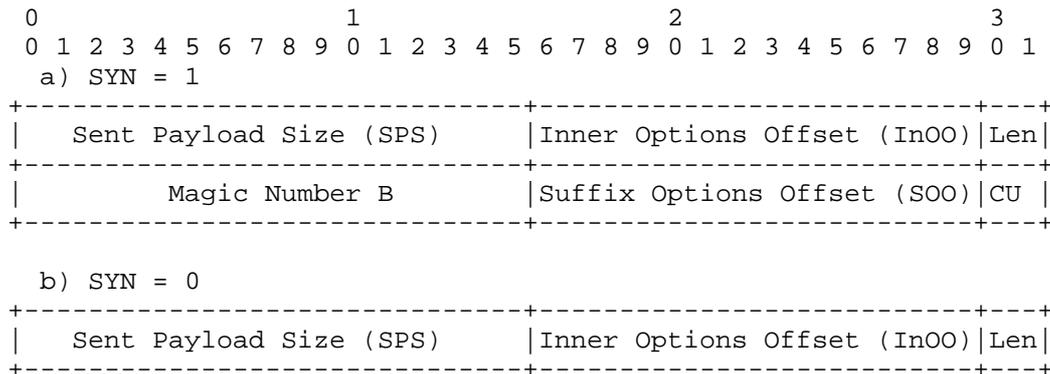
Figure 3: InSpace Option Format

The fields are defined as follows (see Section 3.3 for the rationale
behind these format choices):

Option Length (Len): The 2-bit Len field specifies the length of the
   InSpace Option in 4-octets words (see Section 3.3 for rationale).
   For this experimental specification:

   When SYN=1: the sender MUST use Len=2;

   When SYN=0: the sender MUST use Len=1.

Sent Payload Size (SPS): In this 16-bit field the sender MUST record
   the size in octets of the TCP Payload when it was sent. This
   specification defines the TCP Payload as solely the user-data to
   be passed to the application. This excludes Inner TCP options,
   the InSpace Option and any magic number.

Inner Options Offset (InOO): This 14-bit field defines the total
   size of the Inner TCP Options in 4-octet words.

The following fields are only defined on a segment with SYN=1 (i.e. a
SYN or SYN/ACK):

Magic Number B: The sender MUST fill this 16-bit field with Magic
   Number B {ToDo: Value TBA} to further reduce the chance that a
   receiver will mistake the end of an arbitrary Ordinary Payload for
   the InSpace Option.

Suffix Options Offset (SOO): The 14-bit SOO field defines an
   additional offset in 4-octet words from the start of the Inner
   Options that identifies the extent of the Prefix Options (see
   Section 2.3.2).

Currently Unused (CU): The sender MUST fill the CU field with zeros
   and they MUST be ignored and forwarded unchanged by other nodes,
   even if their value is different.

2.3. Inner TCP Option Processing

2.3.1. Writing Inner TCP Options

2.3.1.1. Constraints on TCP Fast Open

   If an Upgraded TCP Client uses a TCP Fast Open (TFO) cookie
   [I-D.ietf-tcpm-fastopen] in an Upgraded SYN-U, it MUST place the TFO
   option within the Inner TCP Options, beyond the Data Offset.

   This rule is specific to TFO, but it can be generalised to any
   capability similar to TFO as follows: An Upgraded TCP Client MUST NOT
   place any TCP option in the Outer TCP Options of a SYN if it might
   cause a TCP server to pass user-data directly to the application
   before its own 3-way handshake completes.

   If a client uses TCP Fast Open cookies on both the parallel
   connection attempts of a dual handshake, an Upgraded Server will
   deliver the TCP Payload to the application twice before the client
   aborts the Ordinary Connection. This is not a problem, because
   [I-D.ietf-tcpm-fastopen] requires that TFO is only used for
   applications that are robust to duplicate requests.

2.3.1.2. Option Alignment

   If the end of the last Inner TCP Option does not align on a 4-octet
   boundary, the sender MUST append sufficient no-op TCP options. On a
   SYN=1 segment, the end of the Prefix TCP Options MUST be similarly
   aligned.

   If a block-mode transformation (e.g. compression or encryption) is
   being used, the sender might have to add some padding options to
   align the end of the Inner Options with the end of a block. Any
   future encryption specification will need to carefully define this
   padding in order not to weaken the cipher.

2.3.1.3. Sequence Space Coverage

   TCP's sequence number and acknowledgement number space MUST include
   all the TCP Data, i.e. the InSpace Option, any Inner Options, and any
   magic number as well as the TCP Payload. Similarly, the sender MUST
   NOT transmit any form of TCP Data unless the advertised receive
   window is sufficient. These rules have significant implications,
   which are discussed in Section 3.2.4.

2.3.1.4. Presence or Absence of Payload

   Whenever the sender includes non-zero user-data payload in a segment,
   it MUST also include an InSpace Option, whether or not there are any
   Inner Options.

   If the sender includes no user-data in a segment (e.g. pure ACKs,
   RSTs) it MAY include an InSpace Option but it does not have to.
   {ToDo: Consider whether there is any reason to preclude Inner Options
   on a RST, FIN or FIN-ACK.}

   Once a sender has included the InSpace Option and possibly other
   Inner Options on a segment with no TCP Payload, while it has no
   further user-data to send it SHOULD NOT repeat the same set of
   control options on subsequent segments. Thus, in a sequence of pure
   ACKs, any particular set of Inner Options will only appear once, and
   other pure ACKs will be empty. The only envisaged exception to this
   rule would be infrequent repetition (i.e. tens of minutes to hours)
   of the same control options, which might be necessary to provide a
   heartbeat or keep-alive capability.

2.3.2. Reading Inner TCP Options

   The rules for reading Inner TCP Options are divided between the
   following two subsections, depending on whether SYN=1 or SYN=0.

2.3.2.1. Reading Inner TCP Options (SYN=1)

   This subsection applies when TCP receives a segment with SYN=1, i.e.
   when the server receives a SYN or the client receives a SYN/ACK.

   Before processing any TCP options, unless the size of the TCP Data is
   less than 8 octets, an Upgraded Receiver MUST determine whether the
   segment is an Upgraded Segment by checking that all the following
   conditions apply:

   o The first 4 octets of the segment match Magic Number A;

   o The value of the Length field of the InSpace Option is 2;

   o The value of Magic Number B in the InSpace Option is correct;

   o The value of the Sent Payload Size matches the size of the TCP
      Payload.

   If all these conditions pass, the receiver MAY walk the sequence of
   Inner TCP Options, using the length of each to check that the sum of
   their lengths equals InOO. The receiver then concludes that the
   received segment is an Upgraded Segment.

   The receiver then processes the TCP Options in the following order:

   1. Any Prefix TCP options (PrefixOpts in Figure 2)

   2. Any Outer TCP options (OuterOpts in Figure 2);

   3. Any Suffix TCP options (SuffixOpts in Figure 2)

   The receiver removes the magic number, the InSpace Option and each
   TCP Option from the TCP Data as it processes each. This frees up
   receive buffer, so the receiver increases its local value of the
   receive window accordingly. Once only the TCP Payload remains, the
   receiver holds it ready to pass to the application. It then returns
   the appropriate Upgraded Acknowledgement to progress the dual
   handshake (see Section 2.1.1).

   If any of the above tests to find the InSpace Option fails:

   1. the receiver concludes that the received segment is an Ordinary
       Segment. It MUST then proceed by processing any Outer TCP
       options in the TCP Header in the normal order (OuterOpts in
       Figure 2).

2. If some previous control message causes the TCP receiver to alter
    the TCP Data (e.g. decompression, decryption), it reruns the
    above tests to check if the altered TCP Data now looks like an
    Upgraded Segment.

3. If it finds an InSpace Option, it suspends processing the Outer
    TCP Options and instead processes and removes TCP Options in the
    following order:

    1. Any Prefix Inner Options;

    2. Any remaining Outer TCP Options;

    3. Any Suffix Inner Options.

4. If it does not find an InSpace Option, it continues processing
    the remaining Outer TCP Options as normal.

For the avoidance of doubt the above rules imply that, as long as an
InSpace Option has not been found in the segment, the receiver might
rerun the tests for it multiple times if multiple Outer TCP Options
alter the TCP Data. However, once the receiver has found an InSpace
Option, it MUST NOT rerun the tests for an Upgraded Segment in the
same segment.

If the receiver has not found an InSpace Option after processing all
the Outer Options, it returns the appropriate Ordinary
Acknowledgement to progress the dual handshake (see Section 2.1.1).
As normal, it holds any TCP Payload ready to pass to the application.

2.3.2.2. Reading Inner TCP Options (SYN=0)

This subsection applies once the TCP connection has successfully
negotiated to use the upgraded InSpace structure.

As each segment with SYN=0 arrives, the receiver immediately
processes any Outer TCP options.

As the receiver buffers TCP Data, it uses TCP's regular mechanisms to
fill any gaps due to reordering or loss so that it can work its way
along the ordered byte-stream. As the receiver encounters each set
of Inner Options, it MUST process them in the order they were sent,
as illustrated in Figure 4a) in Section 3.2.4. The receiver MUST
remove the InSpace Option and Inner TCP Options from the TCP Data as
it processes them, adding to the receive window accordingly. Once
only the TCP Payload remains the receiver passes it to the
application.

It uses each InSpace Option to calculate the extent of the associated Inner Options (using InOO), and the amount of payload data before the next InSpace Option (using Sent Payload Size). The receiver MUST NOT locate InSpace Options by assuming there is one at the start of the TCP Data in every segment, because resegmentation might invalidate this assumption.

Therefore, the receiver processes the Inner Options in the order they were sent, which is not necessarily the order in which they are received. And if an Inner Option applies to the data stream, the receiver applies it at the point in the data stream where the sender inserted it. As a consequence, the receiver always processes the Inner Options after the Outer Options.

The Inner Options are deliberately placed within the byte-stream so that the sender can transform them along with the payload data, e.g. to compress or encrypt them. A previous control message might have required the TCP receiver to alter the byte-stream before passing it to the application, e.g. decompression or decryption. If so, the TCP receiver applies transformations progressively, to one sent segment at a time, in the following order:

1. The receiver MUST apply any transformations to the byte-stream up to the end of the next set of Inner Options, i.e. over the extent of the next Sent Payload Size, InSpace Option and any Inner Options.

2. The receiver MUST then process and remove the InSpace Option and any Inner Options (which might change the way it transforms the next segment, e.g. a rekey option).

3. Having established the extent of the next sent segment, The receiver returns to step 1.

2.3.3. Forwarding Inner TCP Options

   Middleboxes exist that process some aspects of the TCP Header. Although the present specification defines a new location for Inner TCP Options beyond the Data Offset, this is intended for the exclusive use of the destination TCP implementation. Therefore:

   o A middlebox MUST treat any octets beyond the Data Offset as immutable user-data. Legacy Middleboxes already do not expect to find options beyond the Data Offset anyway.

   o A middlebox MUST NOT defer data in a segment with SYN=1 to a subsequent segment.

A TCP implementation is not necessarily aware whether it is deployed in a middlebox or in a destination, e.g. a split TCP connection might use a regular off-the-shelf TCP implementation. Therefore, a general-purpose TCP that implements the present specification will need a configuration switch to disable any search for options beyond the Data Offset and to enable immediate forwarding of data in a SYN.

2.4. Exceptions

{ToDo: Define behaviour of forwarding or receiving nodes if the structure or format of an Upgraded Segment is not as specified.}

If an Upgraded TCP Receiver receives an InSpace Option with a Length it does not recognise as valid, it MUST drop the packet and acknowledge the octets up to the start of the unrecognised option.

Values of Sent Payload Size greater than $2^{16} - 25$ (=65,511) octets in a regular (non-jumbo) InSpace Option MUST be treated as the distance to the next InSpace option, but they MUST NOT be taken as indicative of the size of the TCP Payload when it was sent. This is because the TCP Payload in a regular IPv6 packet cannot be greater than ($2^{16} -1 - 20 - 4$) octets (given the minimum TCP header is 20 octets and the minimum InSpace Option is 4 octets). A Sent Payload Size of 0xFFFF octets MAY be used to minimise the occurrence of empty InSpace options without permanently disabling the Inner Space protocol for the rest of the connection.

If the size of the payload is greater than 65,511 octets, the sender MUST use a Jumbo InSpace Option (Appendix A.3).

2.5. SYN Flood Protection

An implementation of the Inner Space protocol MUST support the EchoCookie TCP option [I-D.briscoe-tcpm-echo-cookie]. To indicate its support for EchoCookie, an Ordinary Client would send an empty EchoCookie TCP option on the SYN. Support for the Inner Space protocol makes this redundant. Therefore an Inner Space client MUST NOT send an empty EchoCookie TCP option on a SYN-U.

The EchoCookie TCP option replaces the SYN Cookie mechanism [RFC4987], which only has sufficient space to hold the result of one TCP option negotiation (the MSS), and then only a subset of the possible values (see the discussion under Security Considerations Section 6).

3. Design Rationale

    This section is informative, not normative.

3.1. Dual Handshake and Migration to Single Handshake

    In traditional [RFC0793] TCP, the space for options is limited to 40B
    by the maximum possible Data Offset. Before a TCP sender places
    options beyond that, it has to be sure that the receiver will
    understand the upgraded protocol, otherwise it will confuse and
    potentially crash the application by passing it TCP options as if
    they were payload data.

    The Dual Handshake (Section 2.1.1) ensures that a Legacy TCP Server
    will never pass on TCP options as if they were user-data. If a SYN
    carries TCP Data, a TCP server typically holds it back from the
    application until the 3-way handshake completes. This gives the
    client the opportunity to abort the Upgraded Connection if the
    response from the server shows it does not recognise an Upgraded SYN.

    The strategy of sending two SYNs in parallel is not essential to the
    Alternative SYN approach. It is merely an initial strategy that
    minimises latency when the client does not know whether the server
    has been upgraded. Evolution to a single SYN with greater option
    space could proceed as follows:

    o Clients could maintain a white-list of upgraded servers discovered
       by experience and send just the Upgraded SYN-U in these cases.

    o Then, for white-listed servers, the client could send an Ordinary
       SYN only in the rare cases when an attempt to use an Upgraded
       Connection had previously failed (perhaps a mobile client
       encountering a new blockage on a new path to a server that it had
       previously accessed over a good path).

    o In the longer term, once it can be assumed that most servers are
       upgraded and the risk of having to fall back to legacy has dropped
       to near-zero, clients could send just the Upgraded SYN first,
       without maintaining a white-list, but still be prepared to send an
       Ordinary SYN in the rare cases when that might fail.

    There is concern that, although dual handshake approaches might well
    eventually migrate to a single handshake, they do not scale when
    there are numerous choices to be made simultaneously. For instance:

    o trying IPv6 then IPv4 [RFC6555];

   o   and trying SCTP and TCP in parallel
       [I-D.wing-tsvwg-happy-eyeballs-sctp];

   o   and trying ECN and non-ECN in parallel;

   o   and so on.

   Nonetheless, it is not necessary to try every possible combination of
   N choices, which would otherwise require 2^N handshakes (assuming
   each choice is between two options). Instead, a selection of the
   choices could be attempted together. At the extreme, two handshakes
   could be attempted, one with all the new features, and one without
   all the new features.

3.2. In-Band Inner Option Space

3.2.1. Non-Deterministic Magic Number Approach

   This section justifies the magic number approach by contrasting it
   with a more 'conventional' approach. A conventional approach would
   use a regular (Outer) TCP option to point to the dividing line within
   the TCP Data between the extra Inner Options and the TCP Payload.

   This 'conventional' approach cannot provide extra option space over a
   path on which a middlebox strips TCP options that it does not
   recognise. [Honda11] quantifies the prevalence of such paths. It
   reports on experiments conducted in 2010-2011 that found unknown
   options were stripped from the SYN-SYN/ACK exchange on 14% of paths
   to port 80 (HTTP), 6% of paths to port 443 (HTTPS) and 4% of paths to
   port 34343 (unassigned). Further analysis found that the option-
   stripping middleboxes fell into two main categories:

   o   about a quarter appeared to actively remove options that they did
       not recognise (perhaps assuming they might be indicative of an
       attack?);

   o   the rest were some type of higher layer proxy that split the TCP
       connection, unwittingly failing to pass unknown options between
       the two connections.

   In contrast, the magic number approach ensures that not only are the
   Inner Options tucked away beyond the Data Offset, but the option that
   gives the extent of the Inner Options is also beyond the Data Offset
   (see Section 2.2.1). This ensures that all the TCP Headers and
   options up to the Data Offset are completely indistinguishable from
   an Ordinary Segment. It is very unusual for a middlebox not to
   forward TCP Data unchanged, so it will be highly likely (but not
   certain--see Appendix A.2.4) to forward the extra Inner Options.

The downside of the magic number approach is that it is slightly non-deterministic, quantified as follows:

o The probability that an Upgraded SYN=1 segment will be mistaken for an Ordinary Segment is precisely zero.

o In the currently common case of a SYN with zero payload, the probability that it will be mistaken for an Upgraded Segment is also precisely zero.

o However, there will be a very small probability (roughly 2^{-66} or 1 in 74 billion billion ($74 * 10^{18}$)) that payload data in an Ordinary SYN=1 segment could be mistaken for an Upgraded SYN or SYN/ACK, if it happens to contain a pattern in exactly the right place that matches the correct Sent Payload Size, Length and Magic Numbers of an InSpace Option. {ToDo: Estimate how often a collision will occur globally. Rough estimate: 1 connection collision globally every 40 years.}

The above probability is based on the assumptions that:

o the magic numbers will be chosen randomly (in reality they will not--for instance, a magic number that looked just like the start of an HTTP connection would be rejected)

o data at the start of Ordinary SYN=1 segments is random (in reality it is not--the first few bytes of most payloads are very predictable).

Therefore even though 2^{-66} is a vanishingly small probability, the actual probability of a collision will be much lower.

If a collision does occur, it will result in TCP removing a number of 32-bit words of data from the start of a byte-stream before passing it to the application.

3.2.2. Non-Goal: Security Middlebox Evasion

The purpose of locating control options within the TCP Data is not to evade security. Security middleboxes can be expected to evolve to examine control options in the new inner location. Instead, the purpose is to traverse middleboxes that block new TCP options unintentionally--as a side effect of their main purpose--merely because their designers were too careless to consider that TCP might evolve. This category of middleboxes tends to forward the TCP Payload unaltered.

By sitting within the TCP Data, the Inner Space protocol should
traverse enough existing middleboxes to reach critical mass and prove
itself useful. In turn, this will open an opportunity to introduce
integrity protection for the TCP Data (which includes Inner Options).
Whereas today, no operating system would introduce integrity
protection of Outer TCP options, because in too many cases it would
fail and abort the connection. Once the integrity of Inner Options
is protected, it will raise the stakes. Any attempt to meddle with
control options within the TCP Data will not just close off the
theoretical potential benefit of a protocol advance that no-one knows
they want yet; it will fail integrity checks and therefore completely
break any communication. It is unlikely that a network operator will
buy a middlebox that does that.

Then middlebox designers will be on the back foot. To completely
block communications they will need a sound justification. If they
block an attack, that will be fine. But if they want to block
everything abnormal, they will have to block the whole communication,
or nothing. So the operator will want to choose middlebox vendors
who take much more care to ensure their policies track the latest
protocol advances--to avoid costly support calls.

3.2.3. Avoiding the Start of the First Two Segments

Some middleboxes discard a segment sent to a well-known port
(particularly port 80) if the TCP Data does not conform to the
expected app-layer protocol (particularly HTTP). Often such
middleboxes only parse the start of the app-layer header (e.g. Web
filters only continue until they find the URL being accessed, or DPI
boxes only continue until they have identified the application-layer
protocol).

The segment structure defined in Section 2.2.1 would not traverse
such middleboxes. An alternative segment structure that avoids the
start of the first two segments in each direction is defined in
Appendix A.4. It is not mandatory to implement in the present
specification. However, it is hoped that it will be included in some
experimental implementations so that it can be decided whether it is
worth making mandatory.

3.2.4. Control Options Within Data Sequence Space

Including Inner Options within TCP's sequence space gives the sender
a simple way to ensure that control options will be delivered
reliably and in order to the remote TCP, even if the control options
are on segments without user-data. By using TCP's existing stream
delivery mechanisms, it adds no extra protocol processing, no extra
packets and no extra bits.

The sender can even choose to place control options on a segment
without user-data, e.g. to reliably re-key TCP-level encryption on a
connection currently sending no data in one direction. The sender
can even add an InSpace Option without further Inner Options. Then
it can ensure that the segment will automatically be delivered
reliably and in order to the remote TCP, even though it carries no
user-data or other TCP control options, e.g. for a test probe, a
tail-loss probe or a keep-alive.

Figure 4a) illustrates control options arriving reliably and in order
at the receiving TCP stack in comparison with the traditional
approach shown in Figure 4b), in which control options are outside
the sequence space. In the traditional approach, during a period
when the remote TCP is sending no user-data, the local TCP may
receive control options E, B and D without ever knowing that they are
out of order, and without ever knowing that C is missing.

```
a)                   __ ____ _____ _                      __
                    |__|____|_____|_|                    |__|    control
                    :E : D  :  C    :B:                    :A :
          _____:  :     :      : :_____:  :
         |_____|                     |_____|    data

b)                   __
                    |__|  E
                    |_|__  B
                    |____|D                    __
                     \  /                     |__|A    control
                      \/                       \ /
          _____\/_____\/
         |_____||_____|    data
                        !
                     !drop
                 ____!__
                |_____|C
```
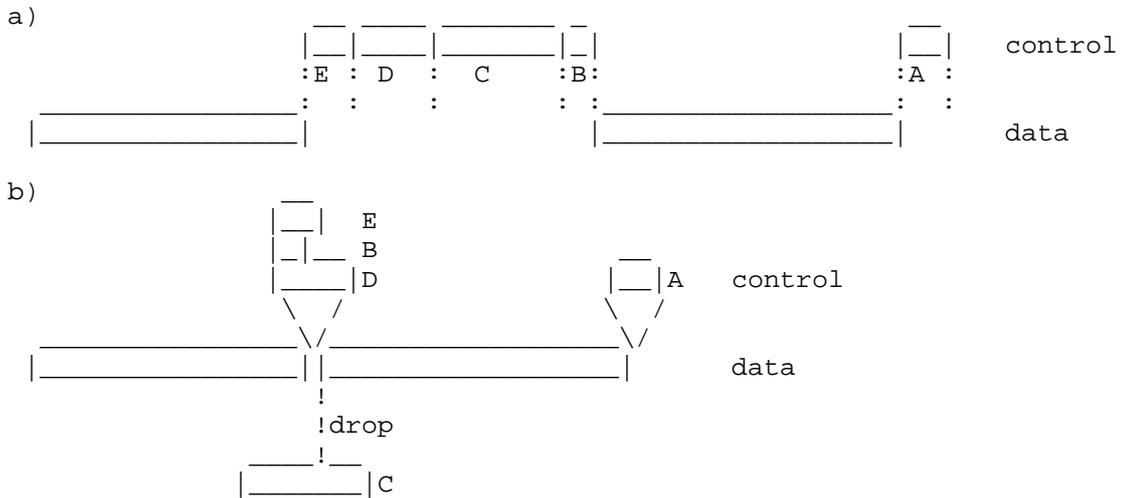
                Figure 4: Control options a) inside vs. b) outside TCP sequence
                                        space`

By including Inner Options within the sequence space, each control
option is automatically bound to the start of a particular byte in
the data stream, which makes it easy to switch behaviour at a
specific point mid-stream (e.g. re-keying or switching to a different
control mode). With traditional TCP options, a bespoke reliable and
ordered binding to the data stream would have to be developed for
each TCP option that needs this capability (e.g. co-ordinating use
of new keys in TCP-AO [RFC5925] or tcpcrypt [I-D.bittau-tcpinc]).

Including Inner Options in sequence also allows the receiver to tell
the sender the exact point at which it encountered an unrecognised
TCP option using only TCP's pre-existing byte-granularity
acknowledgement scheme.

Middleboxes exist that rewrite TCP sequence and acknowledgement
numbers, and they also rewrite options that refer to sequence numbers
(at least those known when the middlebox was produced, such as SACK,
but not any introduced afterwards). If Inner Options were not
included in sequence, the number of bytes beyond the TCP Data Offset
in each segment would not match the sequence number increment between
segments. Then, such middleboxes could unintentionally corrupt the
user-data and options by 'normalising' sequence or acknowledgement
numbering. Fortunately, including Inner Options in sequence improves
robustness against such middleboxes.

3.2.5. Rationale for the Sent Payload Size Field

A middlebox that splits a TCP connection can coalesce and/or divide
the original segments. Segmentation offload hardware introduces
similar resegmentation. Inclusion of the Sent Payload Size field in
the InSpace Option makes the scheme robust against such
resegmentation.

The Sent Payload Size is not strictly necessary on a SYN (SYN=1,
ACK=0) because a SYN is never resegmented. However, for simplicity,
the layout for a SYN is made the same as for a SYN/ACK. This future-
proofs the protocol against the possibility that SYNs might be
resegmented in future. And it makes it easy to introduce the
alternative segment structure of Appendix A.4 if it is needed.

3.3. Rationale for the InSpace Option Format

The format of the InSpace Option (Figure 3) does not necessarily have
to comply with the RFC 793 format for TCP options, because it is not
intended to ever appear in a sequence of TCP options. In particular,
it does not need an Option Kind, because the option is always in a
known location. In effect the magic number serves as a multi-octet
Option Kind for the first InSpace Option, and the location of each
subsequent options is always known as an offset from the previous
one, using InOO and Sent Payload Size fields.

Other aspects of the layout are justified as follows:

Length: Whatever the size of the InSpace Option, the right-hand edge
   of the Length field is always located 4 octets from the start of
   the option, so that the receiver can find it to determine the
   layout of the rest of the option. The option is always a multiple

of 4 octets long, so that any subsequent Inner TCP Options comply
with TCP's option alignment requirements.

Sent Payload Size: This field is 16 bits wide, which is reasonable
   given segment size cannot exceed the limits set by the Total
   Length field in the IPv4 header and the Payload Length field in
   the IPv6 header, both of which are 16 bits wide.

   If the sender were to use a jumbogram [RFC2675], it could use the
   Jumbo InSpace Option defined in Appendix A.3, which offers a
   32-bit Sent Payload Size field. The Jumbo InSpace Option is not
   mandatory to implement for the present experimental specification.
   Even if it is implemented, it is only defined when SYN=0, given
   use of a jumbogram for a SYN or SYN/ACK would significantly exceed
   other limits that TCP sets for these segments.

InSpace Options Offset The 14-bit field is in units of 4-octet
   words, in order to restrict Inner Options to no less than the size
   of a maximum sized segment (given $4 * 2^{14} = 2^{16}$ octets).

When SYN=1 the layout of the InSpace Option is extended to include:

Suffix Options Offset: The SOO field is the same 14-bit width as the
   InOO field, and for the same reason. Both the SOO and InOO fields
   are aligned 2 bits to the left of a word boundary so that they can
   be used directly in units of octets by masking out the 2-bit field
   to the right.

Magic Number B: The 32-bit size of Magic Number A is not enough to
   reduce the probability of mistaking the start of an Ordinary SYN
   Payload for the start of the Inner Space protocol. A 64-bit magic
   number could have been provided by using the next 4-octet word,
   but this would be unnecessarily large. Therefore, when SYN=1, 16
   more bits of magic number are provided within the InSpace Option.
   Otherwise, these 16-bits would only have to be used for padding to
   align with the next 4-octet word boundary anyway.

3.4. Protocol Overhead

   The overhead of the Inner Space protocol is quantified as follows:

   Dual Handshake:

      Latency:

         Upgraded Server : zero;

         Legacy Server: worst latency of the dual handshakes.

Connection Rate: The typical connection rate will inflate by P*D,
   where:

   P [0-100%] is the proportion of connections that use extra
      option space;

   D [0-100%] is the proportion of these that use a dual
      handshake (the remainder use a single handshake, e.g. by
      caching knowledge of upgraded servers).

   For example, if P=80% and D=10%, the connection rate will
   inflate by 8%. P is difficult to predict. D is likely to be
   small, and in the longer term it should reduce to the
   proportion of connections to remaining legacy servers, which
   are likely to be the less frequently accessed ones. In the
   worst case if both P & D are 100%, the maximum that the
   connection rate can inflate by is 100% (i.e. to twice present
   levels).

Connection State: Connection state on servers and middleboxes
   will inflate by P*D/R, where

   R is the average hold time of connection state measured in
      round trip times

   This is because a server or middlebox only holds dual
   connection state for one round trip, until the RST on one of
   the two connections. For example, keeping P & D as they were
   in the above example, if R = 3 round trips {ToDo: TBA},
   connection state would inflate by 2.7%. In the longer term, any
   extra connection state would be focused on legacy servers, with
   none on upgraded servers. Therefore, if memory for dual
   handshake flow state was a problem, upgrading the server to
   support the Inner Space protocol would solve the problem.

Network Traffic: The network traffic overhead is 2*H*P*D/J
   counting in bytes or 2*P*D/K counting in packets, where

   H is 88B for IPv4 or 108B for IPv6 (assuming the Ordinary SYN
      and SYN/ACK have a TCP header packed to the maximum of 60B
      with TCP options, they have no TCP payload, their IP headers
      have no extensions and the InSpace Option in the SYN-U and
      SYN/ACK-U is 8B);

   J is the average number of bytes per TCP connection (in both
      directions)

K   is the average number of packets per TCP connection (in both
    directions);

For example, keeping and P & D as they were in the above
example, if J = 50KiB for IPv4 and K = 70 packets (ToDo: TBA),
traffic overhead would be 0.03% counting in bytes or 0.2%
counting in packets.

Processing: {ToDo: Implementation tests}

InSpace Option on every non-empty SYN=0 segment:

Network Traffic: The traffic overhead is P*Q*4/F, where

Q   is the proportion of Inner Space connections that leave the
    protocol enabled after the initial handshake;

F   is the average frame size in bytes (assuming one segment per
    frame).

This is because the InSpace option adds 4B per segment. For
example, keeping P as it was in the above example and taking
Q=10% and F=750B, the traffic overhead is 0.04%. It is as
difficult to predict Q as it is to predict P.

Processing: {ToDo: Implementation tests}

4. Interaction with Pre-Existing TCP Implementations

4.1. Compatibility with Pre-Existing TCP Variants

A TCP option MUST by default only be used as an Outer Option, unless
it is explicitly specified that it can (or must) be used as an Inner
Option. The following list of pre-existing TCP options can be
located as Inner Options:

o Maximum Segment Size (MSS) [RFC0793];

o SACK-ok [RFC2018];

o Window Scale [RFC7323];

o Multipath TCP [RFC6824], except the Data ACK part of the Data
   Sequence Signal (DSS) option;

o TCP Fast Open [I-D.ietf-tcpm-fastopen];

o The tcpcrypt CRYPT option [I-D.bittau-tcpinc].

The following MUST NOT be located as Inner Options:

o Timestamp [RFC7323];

o SACK [RFC2018];

o The Data ACK part of the DSS option of Multipath TCP [RFC6824];

o TCP-AO [RFC5925];

o The tcpcrypt MAC option [I-D.bittau-tcpinc] as long as it covers
   the TCP header.

{ToDo: The above list is not authoritative. Many of the above
schemes involve multiple different types of TCP option, and all the
types need to be separately assessed.}

The Inner Space protocol supports TCP Fast Open, by constraining the
client to obey the rules in Section 2.3.1.1).

All the sub-types of the MPTCP option [RFC6824] except one could be
located as Inner Options. That is, MP_CAPABLE, MP_JOIN, ADD_ADDR(2),
REMOVE_ADDR, MP_PRIO, MP_FAIL, MP_FASTCLOSE. The Data Sequence
Signal (DSS) of MPTCP consists of four separable parts: i) the Data
ACK; ii) the mapping between the Data Sequence Number and the Subflow
Sequence Number over a Data-Level Length; iii) the Checksum; and iv)
the DATA_FIN flag. If MPTCP were re-factored to take advantage of
the Inner Space protocol, all these parts except the Data ACK could
be located as Inner Options (the Checksum would not be necessary).

The MPTCP Data ACK has to remain as an Outer Option otherwise there
would be a risk of flow control deadlock, as pointed out in
[Raiciu12]. For instance, a Web client might pipeline multiple
requests that fill a Web server's receive buffer, while the Web
server might be busy sending a large response to the first request
before it reads the second request. If the Data ACK were an Inner
Option, the Web client would have to stop acknowledging the first
response from the server (due to lack of receive window). Then the
server would not be able to move on to the next request--a classic
deadlock.

The TCP-AO has to be located as an Outer Option to prevent the
possibility of flow-control deadlock (because it would consume
receive window on pure ACKs).

All sub-options of the tcpcrypt CRYPT option could be located as
Inner Options. However, as long as the tcpcrypt MAC option covers

the TCP header and Outer Options, it has to be located as an Outer
Option for the same deadlock reason as TCP-AO.

An Upgraded Server can support SYN Cookies [RFC4987] for Ordinary
Connections. For Upgraded Connections Section 2.5 defines a new
EchoCookie TCP option that is a prerequisite for InSpace
implementations, and provides sufficient space for the more extensive
connection state requirements of an InSpace server.

{ToDo: TCP States and Transitions, Connectionless Resets, ICMP
Handling, Forward-Compatibility.}

4.2. Interaction with Middleboxes

The interaction with the assumptions about TCP made by middleboxes is
covered extensively elsewhere:

o Section 2.3.3 specifies forwarding behaviour for Inner Options;

o The following sections explain the Inner Space protocol approach
   to middlebox traversal:

   * Section 3.2.1 justifies the magic number approach;

   * Section 3.2.2 explains why the protocol will remain robust as
      middlboxes evolve;

   * Section 3.2.4 justifies including Inner Options in sequence;

   * Section 3.2.5) explains how the protocol will remain robust to
      resegmentation.

4.3. Interaction with the Pre-Existing TCP API

An aim of the Inner Space protocol is for legacy applications to
continue to just work without modification. Therefore it is expected
that the dual handshaking logic and any maintenance of a cached
white-list of servers that support the Inner Space protocol will be
implemented beneath the well-known socket interface.

Inner Space implementations will need to comply with the following
behaviours to ensure that legacy applications continue to receive
predictable behaviour from the socket interface:

Querying local port (TCP client): If an application calls
   "getsockname()" while the TCP client behind the socket is engaged
   in a dual TCP handshake, the call SHOULD block until the local TCP

has aborted one of the connections so it knows which of the two
ports will continue to be used.

Binding to an explicit port: If an application specifies that it
    wants the TCP client to use a specific port, the Inner Space
    capability MUST be disabled, because the dual handshake has to try
    two ports. Use of a specific port might be necessary, for example
    in a port-testing application or if the application wants to
    explicitly control all the handshaking logic of the Inner Space
    protocol itself.

Logging: The dual handshake will show up as a specific signature in
    logs of network activity. Log formats might not be able to record
    two local ports against one socket, so logs might contain
    unexpected or erroneous data. Even if logs correctly track both
    connection attempts, log analysis software might not expect to see
    one socket attempt to use two different ports. {ToDo: All this
    needs to be turned into a predictability requirement.}

Note that Inner Space has no impact on queries for the remote port
from a TCP server. If an application calls "getpeername()" while the
TCP server behind the socket is (unwittingly) engaged in a dual
handshake, it will return the port of the remote client, even though
this connection might subsequently be aborted. This is because a TCP
server is not aware of whether it is part of a dual handshake.

It would be appropriate to enable the Inner Space protocol on a per-
host or per-user basis. The necessary configuration switch does not
need to be standardised, but it might allow the following three
states:

Enabled: The stack will enable Inner Space on any TCP connection
    that that needs Inner Space for its TCP options. The stack might
    still disable the Inner Space protocol autonomously after the
    initial handshake if it is not needed.

Forwarding: The Forwarding mode is for TCP implementations on
    middleboxes that implement split TCP connections, as discussed in
    Section 2.3.3. Forwarding mode is similar to Disabled, except it
    forwards data in SYN without deferring it until the incoming
    connection is established.

Disabled: Inner Space is not enabled by default on any connections,
    except those that specifically request it.

The socket API might also need to be extended for future applications
that want to control the Inner Space protocol explicitly. Experience

will determine the best API, so these ideas are merely informational
suggestions at this stage:

Enabling/disabling Inner Space: As well as the above per-host or
   per-user switches, the extended API might need to allow an
   application to disable Inner Options on a per-socket basis (e.g.
   for testing). A socket might need to be opened in one of three
   possible Inner Space modes: i) Enabled; ii) Enabled initially but
   can be disabled autonomously by the stack if redundant; iii)
   Enabled initially, then disables itself after the SYN/ACK; and iv)
   Disabled. It also ought to be possible for an application to
   disable Inner Options on-demand mid-connection.

Querying support for Inner Space: An application might need to be
   able to determine whether the host supports Inner Space and in
   which mode it is enabled on a particular socket. For instance, an
   application might need to choose different socket options
   depending on whether Inner Space is enabled to make the necessary
   space available.

Latency vs Efficiency: A socket that prefers efficient use of
   connection state over latency might use the optional explicit
   variant of the dual handshake (Appendix B). It is unlikely that a
   new option specific to Inner Space would be needed to express this
   preference, as many operating systems already offer a similar
   socket option.

Logging: Log formats and log analysis software might need to be
   extended to distinguish between the deliberate RST within the dual
   handshake and an unexpected connection RST.

5. IANA Considerations

   This specification requires IANA to allocate values from the TCP
   Option Kind name-space against the following names:

   o "Inner Option Space Upgraded (InSpaceU)"

   o "Inner Option Space Ordinary (InSpaceO)"

   o "ModeSwitch"

   Early implementation before the IANA allocation MUST follow [RFC6994]
   and use experimental option 254 and respective Experiment IDs:

   o 0xUUUU (16 bits);

   o 0xOOOO (16 bits);

o   0xMMMM (16 bits);

{ToDo: Values TBA and register them with IANA} then migrate to the
assigned option after allocation.

6. Security Considerations

Certain cryptographic functions have different coverage rules for the
TCP Header and TCP Payload. Placing some TCP options beyond the Data
Offset could mean that they are treated differently from regular TCP
options. This is a deliberate feature of the protocol, but
application developers will need to be aware that this is the case.

A malicious host can send bogus SYN segments with a spoofed source IP
address (a SYN flood attack). The Inner Space protocol does not
alter the feasibility of this attack. However, the extra space for
TCP options on a SYN allows the attacker to include more TCP options
on a SYN than before, so it can make a server do more option
processing before replying with a SYN/ACK. To mitigate this problme,
a server under stress could deprioritise SYNs with longer option
fields to focus its resources on SYNs that require less processing.

Each SYN in a SYN flood attack causes a TCP server to consume memory.
The Inner Space protocol allows a potentially large amount of TCP
option state to be negotiated during the SYN exchange, which could
exhaust the TCP server's memory. The EchoCookie TCP option (see
Section 2.5) allows the server to place this state in a cookie and
send it on the SYN/ACK to the purported address of the client--rather
than hold it in memory.

Then, as long as the client returns the cookie on the acknowledgement
and the server verifies it, the server can recover its full record of
all the TCP options it negotiated and continue the connection without
delay. On the other hand, the server's responses to SYNs from
spoofed addresses will scatter to those spoofed addresses and the
server will not have consumed any memory while waiting in vain for
them to reply. See the Security Considerations in
[I-D.briscoe-tcpm-echo-cookie] for how the EchoCookie facility
protects against reflection and amplification attacks.

7. Acknowledgements

The idea of this approach grew out of discussions with Joe Touch
while developing draft-touch-tcpm-syn-ext-opt, and with Jana Iyengar
and Olivier Bonaventure. The idea that it is architecturally
preferable to place a protocol extension within a higher layer, and
code its location into upgraded implementations of the lower layer,
was originally articulated by Rob Hancock. {ToDo: Ref?} The following

people provided useful review comments: Joe Touch, Yuchung Cheng,
John Leslie, Mirja Kuehlewind, Andrew Yourtchenko, Costin Raiciu,
Marcelo Bagnulo Braun, Julian Chesterfield and Jaime Garcia.

8. References

8.1. Normative References

[I-D.ietf-tcpm-fastopen]
          Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP
          Fast Open", draft-ietf-tcpm-fastopen-10 (work in
          progress), September 2014.

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

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

[RFC6994] Touch, J., "Shared Use of Experimental TCP Options", RFC
          6994, August 2013.

8.2. Informative References

[Honda11] Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A.,
          Handley, M., and H. Tokuda, "Is it Still Possible to
          Extend TCP?", Proc. ACM Internet Measurement Conference
          (IMC'11) 181--192, November 2011.

[I-D.bittau-tcpinc]
          Bittau, A., Boneh, D., Hamburg, M., Handley, M., Mazieres,
          D., and Q. Slack, "Cryptographic protection of TCP Streams
          (tcpcrypt)", draft-bittau-tcpinc-01 (work in progress),
          July 2014.

[I-D.briscoe-tcpm-echo-cookie]
          Briscoe, B., "The Echo Cookie TCP Option", draft-briscoe-
          tcpm-echo-cookie-00 (work in progress), October 2014.

[I-D.wing-tsvwg-happy-eyeballs-sctp]
          Wing, D. and P. Natarajan, "Happy Eyeballs: Trending
          Towards Success with SCTP", draft-wing-tsvwg-happy-
          eyeballs-sctp-02 (work in progress), October 2010.

[Iyengar10]
          Iyengar, J., Ford, B., Ailawadi, D., Amin, S., Nowlan, M.,
          Tiwari, N., and J. Wise, "Minion--an All-Terrain Packet
          Packhorse to Jump-Start Stalled Internet Transports",
          Proc. Int'l Wkshp on Protocols for Future, Large-scale &
          Diverse Network Transports (PFLDnet'10) , November 2010.

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

[RFC2675] Borman, D., Deering, S., and R. Hinden, "IPv6 Jumbograms",
          RFC 2675, August 1999.

[RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common
          Mitigations", RFC 4987, August 2007.

[RFC5925] Touch, J., Mankin, A., and R. Bonica, "The TCP
          Authentication Option", RFC 5925, June 2010.

[RFC6555] Wing, D. and A. Yourtchenko, "Happy Eyeballs: Success with
          Dual-Stack Hosts", RFC 6555, April 2012.

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

[RFC7323] Borman, D., Braden, B., Jacobson, V., and R.
          Scheffenegger, "TCP Extensions for High Performance", RFC
          7323, September 2014.

[Raiciu12]
          Raiciu, C., Paasch, C., Barre, S., Ford, A., Honda, M.,
          Duchene, F., Bonaventure, O., and M. Handley, "How Hard
          Can It Be? Designing and Implementing a Deployable
          Multipath TCP", Proc. USENIX Symposium on Networked
          Systems Design and Implementation , April 2012.

Appendix A. Protocol Extension Specifications

   This appendix specifies protocol extensions that are OPTIONAL while
   the specification is experimental. If an implementation includes an
   extension, this section gives normative specification requirements.

However, if the extension is not implemented, the normative
requirements can be ignored.

{Temporary note: The IETF may wish to consider making some of these
extensions mandatory to implement if early testing shows they are
useful or even necessary. Or it may wish to make at least the
receiving side mandatory to implement to ensure that two-ended
experiments are more feasible.}

A.1. Disabling InSpace and Generic Connection Mode Switching

This appendix is normative. It is separated from the body of the
specification because it is OPTIONAL to implement while the Inner
Space protocol is experimental. It defines the new ModeSwitch TCP
option illustrated in Figure 5. This option provides a facility to
disable the Inner Space protocol for the remainder of a connection.
It also provides a general-purpose facility for a TCP connection to
co-ordinate between the endpoints before switching into a yet-to-be-
defined mode.

```
 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
+---------------+---------------+-----------+-+-+
|  ModeSwitch   | Length=3      |Flags (CU) |I|R|
+---------------+---------------+-----------+-+-+
```
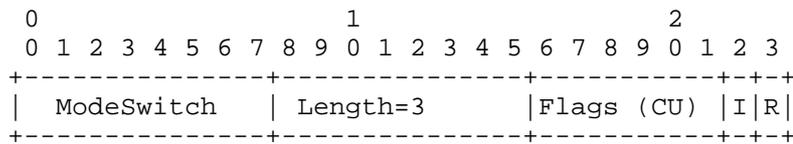
Figure 5: The ModeSwitch TCP Option

The Option Kind is ModeSwitch, the value of which is to be allocated
by IANA {ToDo: Value TBA}. ModeSwitch MUST be used only as an Inner
Option, because it uses the reliable ordered delivery property of
Inner Options. Therefore implementation of the Inner Space protocol
is REQUIRED for an implementation of ModeSwitch. Nonetheless,
ModeSwitch is a generic facility for switching a connection between
yet-to-be-defined modes that do not have to relate to extra option
space.

The sender MUST set the option Length to 3 (octets). The Length
field MUST be forwarded unchanged by other nodes, even if its value
is different.

The Flags field is available for defining modes of the connection.
Only two connection modes are currently defined. The first 6 bits of
the Flags field are Currently Unused (CU) and the sender MUST set
them to zero. The CU flags MUST be ignored and forwarded unchanged
by other nodes, even if their value is non-zero.

The two 1-bit connection mode flags that are currently defined have
the following meanings:

o R: Request flag if 1. Request mode is a special mode that allows
   the hosts to co-ordinate a change to any other mode(s);

o I: Inner Space mode: Enabled if 1, Disabled if 0.

The default Inner Space mode at the start of a connection is I=1,
meaning Inner Space is in enabled mode.

The procedure for changing a mode or modes is as follows:

o The host that wants to change modes (the requester) sends a
   ModeSwitch message as an Inner Option with R=1 and with the other
   flag(s) set to the mode(s) it wants to change to. The requester
   does not change modes yet.

o The responder echoes the mode flag(s) it is willing to change to,
   with the request flag R=0.

o The half-connection from the responder changes to the mode(s) it
   confirms directly after the end of the segment that echoes its
   confirmation, i.e. after the last octet of the TCP Payload
   following the ModeSwitch option that echoes its confirmation.
   Therefore it sends the segment carrying the confirmation in the
   prior mode(s) of the connection.

o Once the requester receives the responder's confirmation message,
   it re-echoes its confirmation of the responder's confirmation,
   with the mode(s) set to those that both hosts agree on and R=0.

o The half-connection from the requester changes to the mode(s) it
   confirms directly after the end of the segment that re-echoes its
   confirmation. Therefore it sends the segment carrying the
   confirmation in the prior mode(s) of the connection.

o The responder can refuse a request to change into a mode in any
   one of three ways:

   * either implicitly by never confirming it;

   * or explicitly by sending a message with R=0 and the opposite
      mode;

   * or explicitly be sending a counter-request to switch to the
      opposite mode (that the connection is already in) with R=1.

The regular TCP sequence numbers and acknowledgement numbers of
requests or confirmations can be used to disambiguate overlapping
requests or responses.

Once a host switches to Disabled mode, it MUST NOT send any further
InSpace Options. Therefore it can send no further Inner Options and
it cannot switch back to Enabled mode for the rest of the connection.

To temporarily reduce InSpace overhead without permanently disabling
the protocol, the sender can use a value of 0xFFFF in the Sent
Payload Size (see Section 2.4).

A.2. Dual Handshake: The Explicit Variant

This appendix is normative. It is separated from the body of the
specification because it is OPTIONAL to implement while the Inner
Space protocol is experimental. It is not mandatory to implement
because it will be more useful once the Inner Space protocol has
become accepted widely enough that fewer middleboxes will discard SYN
segments carrying this option (see Appendix B for when best to deploy
it). It only works if both ends support it, but it can be deployed
one end at a time, so there is no need for support in early
experimental implementations.

{Temporary note: The choice between the explicit handshake in the
present section or the handshake in Section 2.1.1 is a tradeoff
between robustness against middlebox interference and minimal server
state. During the IETF review process, one might be chosen as the
only variant to go forward, at which point the other will be deleted.
Alternatively, the IETF could require a server to understand both
variants and a client could be implemented with either, or both. If
both, the application could choose which to use at run-time. Then we
will need a section describing the necessary API.}

This explicit dual handshake is similar to that in Section 2.1.1,
except the SYN that the Upgraded Client sends on the Ordinary
Connection is explicitly distinguishable from the SYN that would be
sent by a Legacy Client. Then, if the server actually is an Upgraded
Server, it can reset the Ordinary Connection itself, rather than
creating connection state for at least a round trip until the client
resets the connection.

For an explicit dual handshake, the TCP client still sends two
alternative SYNs: a SYN-O intended for Legacy Servers and a SYN-U
intended for Upgraded Servers. The two SYNs MUST have the same
network addresses and the same destination port, but different source
ports. Once the client establishes which type of server has
responded, it continues the connection appropriate to that server

type and aborts the other. The SYN intended for Upgraded Servers
includes additional options within the TCP Data (the SYN-U defined as
before in Section 2.2.1).

Table 2 summarises the TCP 3-way handshake exchange for each of the
two SYNs in the two right-hand columns, between an Upgraded TCP
Client (the active opener) and either:

1. a Legacy Server, in the top half of the table (steps 2-4), or

2. an Upgraded Server, in the bottom half of the table (steps 2-4)

The table uses the same layout and symbols as Table 1, which has
already been explained in Section 2.1.1.

```
+------+----------------+-------------------+-------------------+
|      |                | Ordinary          | Upgraded          |
|      |                | Connection        | Connection        |
+------+----------------+-------------------+-------------------+
| 1    | Upgraded Client| >SYN-O            | >SYN-U            |
|      |                |                   |                   |
| /\/\ | /\/\/\/\/\/\/\ | /\/\/\/\/\/\/\/\ | /\/\/\/\/\/\/\/\ |
| 2    |  Legacy Server | <SYN/ACK          | <SYN/ACK          |
|      |                |                   |                   |
| 3a   | Upgraded Client| Waits for response|                   |
|      |                | to both SYNs      |                   |
|      |                |                   |                   |
| 3b   |       "        | >ACK              | >RST              |
|      |                |                   |                   |
| 4    |                | Cont...           |                   |
|      |                |                   |                   |
| /\/\ | /\/\/\/\/\/\/\ | /\/\/\/\/\/\/\/\ | /\/\/\/\/\/\/\/\ |
| 2    | Upgraded Server| <RST              | <SYN/ACK-U        |
|      |                |                   |                   |
| 3    | Upgraded Client|                   | >ACK              |
|      |                |                   |                   |
| 4    |                |                   | Cont...           |
+------+----------------+-------------------+-------------------+
```

       Table 2: Explicit Variant of Dual 3-Way Handshake in Two Server
                               Scenarios

As before, an Upgraded Server MUST respond to a SYN-U with a SYN/ACK-
U. Then, the client recognises that it is talking to an Upgraded
Server.

Unlike before, an Upgraded Server MUST respond to a SYN-O with a RST.
However, the client cannot rely on this behaviour, because a

middlebox might be stripping Outer TCP Options which would turn the
SYN-O into a regular SYN before it reached the server. Then the
handshake would effectively revert to the implicit variant.
Therefore the client's behaviour still depends on which SYN-ACK
arrives first, so its response to SYN-ACKs has to follow the rules
specified for the implicit handshake variant in Section 2.1.1.

The rules for processing TCP options are also unchanged from those in
Section 2.3.

A.2.1. SYN-O Structure

The SYN-O is merely a SYN with an extra InSpaceO Outer TCP Option as
shown in Figure 6. It merely identifies that the SYN is opening an
Ordinary Connection, but explicitly identifies that the client
supports the Inner Space protocol.

```
 0                   1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+---------------+---------------+
| Kind=InSpaceO | Length=2      |
+---------------+---------------+
```


                  Figure 6: An InSpaceO TCP Option Flag

An InSpaceO TCP Option has Option Kind InSpaceO with value {ToDo:
Value TBA} and MUST have Length = 2 octets.

To use this option, the client MUST place it with the Outer TCP
Options. A Legacy Server will just ignore this TCP option, which is
the normal behaviour for an option that TCP does not recognise
[RFC0793].

A.2.2. Retransmission Behaviour - Explicit Variant

If the client receives a RST on one connection, but a short while
after that {ToDo: duration TBA} the response to the SYN-U has not
arrived, it SHOULD retransmit the SYN-U. If latency is more
important than the extra TCP option space, in parallel to any
retransmission, or instead of any retransmission, the client MAY send
a SYN without any InSpace TCP Option, in case this is the cause of
the black-hole. However, the presence of the RST implies that the
SYN with the InSpaceO TCP Option (the SYN-O) probably reached the
server, therefore it is more likely (but not certain) that the lack
of response on the other connection is due to transmission loss or
congestion loss.

If the client receives no response at all to either the SYN-O or the
SYN-U, it SHOULD solely retransmit one or the other, not both. If
latency is more important than the extra TCP option space, it SHOULD
send a SYN without an InSpaceO TCP Option. Otherwise it SHOULD
retransmit the SYN-U. It MUST NOT retransmit both segments, because
the lack of response could be due to severe congestion.

A.2.3. Corner Cases

There is a small but finite possibility that the Explicit Dual
Handshake might encounter the cases below. The Implicit Handshake
(Section 2.1.1) is robust to these possibilities, but the Explicit
Handshake is not, unless the following additional rules are followed:

Both successful: This could occur if one load-sharing replica of a
   server is upgraded, while another is not. This could happen in
   either order but, in both cases, the client aborts the last
   connection to respond:

   * The client completes the Ordinary Handshake (because it
      receives a SYN/ACK), but then, before it has aborted the
      Upgraded Connection, it receives a SYN/ACK-U on it. In this
      case, the client MUST abort the Upgraded Connection even though
      it would work. Otherwise the client will have opened both
      connections, one with Inner TCP Options and one without. This
      could confuse the application.

   * The client completes the Upgraded Connection after receiving a
      SYN/ACK-U, but then it receives a SYN/ACK in response to the
      SYN-O. In this case, the client MUST abort the connection it
      initiated with the SYN-O.

Both aborted: The client might receive a RST in response to its SYN-
   O, then an Ordinary SYN/ACK on its Upgraded Connection in response
   to its SYN-U. This could occur i) if a split connection middlebox
   actively forwards unknown options but holds back or discards data
   in a SYN; or ii) if one load-sharing replica of a server is
   upgraded, while another is not.

   Whatever the likely cause, the client MUST still respond with a
   RST on its Upgraded Connection. Otherwise, its Inner TCP Options
   will be passed as user-data to the application by a Legacy Server.

   If confronted with this scenario where both connections are
   aborted, the client will not be able to include extra options on a
   SYN, but it might still be able to set up a connection with extra
   option space on all the other segments in both directions using
   the approach in Appendix A.2.4. If that doesn't work either, the

       client's only recourse is to retry a new dual handshake on
       different source ports, or ultimately to fall-back to sending an
       Ordinary SYN.

A.2.4. Workround if Data in SYN is Blocked

    If a path either holds back or discards data in a SYN-U, but there is
    evidence that the server is upgraded from a RST response to the SYN-
    O, the strategy below might at least allow a connection to use extra
    option space on all the segments except the SYN.

    It is assumed that the symptoms described in the 'both aborted' case
    (Appendix A.2.3) have occurred, i.e. the server has responded to the
    SYN-O with a RST, but it has responded to the SYN-U with an Ordinary
    SYN/ACK not a SYN/ACK-U, so the client has had to RST the Upgraded
    Connection as well. In this case, the client SHOULD attempt the
    following (alternatively it MAY give up and fall back to opening an
    Ordinary TCP connection).

    The client sends an 'Alternative SYN-U' by including an InSpaceU
    Outer TCP Option (Figure 7). This Alternative SYN-U merely flags
    that the client is attempting to open an Upgraded Connection. The
    client MUST NOT include any Inner Options or InSpace Option or Magic
    Number. If the previous aborted SYN/ACK-U acknowledged the data that
    the client sent within the original SYN-U, the client SHOULD resend
    the TCP Payload data in the Alternative SYN-U, otherwise it might as
    well defer it to the first data segment.

    0   1
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
    +---------------+---------------+
    | Kind=InSpaceU | Length=2 |
    +---------------+---------------+

                 Figure 7: An InSpaceU Flag TCP option

    An InSpaceU Flag TCP Option has Option Kind InSpaceU with value
    {ToDo: Value TBA} and MUST have Length = 2 octets.

    To use this option, the client MUST place it with the Outer TCP
    Options. A Legacy Server will just ignore this TCP option, which is
    the normal behaviour for an option that TCP does not recognise
    [RFC0793]. Because the client has received a RST from the server in
    response to the SYN-O it can assume that the server is upgraded. So
    the client probably only needs to send a single Alternative SYN-U in
    this repeat attempt. Nonetheless, the RST might have been spurious.

Therefore the client MAY also send an Ordinary SYN in parallel, i.e.
using the Implicit Dual Handshake (Section 2.1.1).

If an Upgraded Server receives a SYN carrying the InSpaceU option, it
MUST continue the rest of the connection as if it had received a full
SYN-U (Section 2.2), i.e. by processing any Outer Options in the
SYN-U and responding with a SYN/ACK-U.

A.3. Jumbo InSpace TCP Option (only if SYN=0)

This appendix is normative. It is separated from the body of the
specification because it is OPTIONAL to implement while the Inner
Space protocol is experimental. In experimental implementations, it
will be sufficient to implement the required behaviour for when the
Length of a received InSpace Option is not recognised (Section 2.4).

If the IPv6 Jumbo extension header is used, the SentPayloadSize field
will need to be 4 octets wide, not 2 octets. This section defines
the format of the InSpace Option necessary to support jumbograms.

If sending a jumbogram, a sender MUST use the InSpace Option format
defined in Figure 8. All the fields have the same meanings as
defined in Section 2.2.2, except InOO and SentPayloadSize use more
bits.

When reading a segment, the Jumbo InSpace Option could be present in
a packet that is not a jumbogram (e.g. due to resegmentation).
Therefore a receiver MUST use the Jumbo InSpace Option to work along
the stream irrespective of whether arriving packets are jumbo sized
or not.

```
 0                   1                   2                   3
 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
+------------------------------------------------------------+---+
|                 Inner Options Offset (InOO)                |Len|
+------------------------------------------------------------+---+
|                   Sent Payload Size (SPS)                      |
+----------------------------------------------------------------+
```
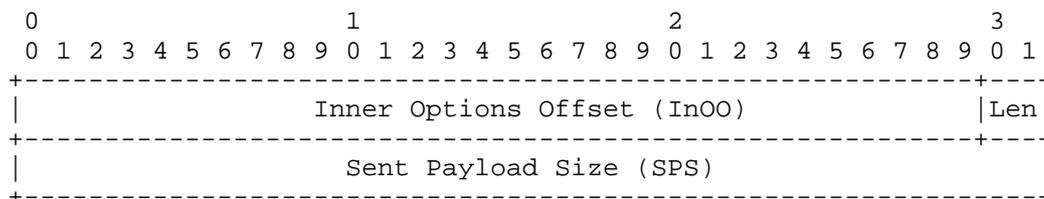
                Figure 8: InSpace Option for a Jumbo Data-UNJH

A.4. Upgraded Segment Structure to Traverse DPI boxes

This appendix is normative. It is separated from the body of the
specification because it is OPTIONAL to implement while the Inner
Space protocol is experimental. If a receiver has implemented the
Inner Space protocol but not this extension, no mechanism is provided
for it to ask the sender to fall-back to the base Inner Space

protocol if it is sent a segment formatted according to this
extension. However, it will at least fall-back naturally to regular
TCP behaviour because of the dual handshake.

In experiments conducted between 2010 and 2011, [Honda11] reported
that 7 of 142 paths (about 5%) blocked access to port 80 if the
payload was not parsable as valid HTTP. This variant of the
specification has been defined in case experiments prove that it
significantly improves traversal of such deep packet inspection (DPI)
boxes.

This variant starts the TCP Data with the expected app-layer headers
on the first two segments in each direction:

SYN=1: The structure in Figure 9a) is used on a SYN or SYN/ACK. The
    sender locates the 4-octet Magic Number A at the end of the
    segment. The sender right-aligns the 8-octet InSpace Option just
    before Magic Number A. Then it right-aligns the Inner Options
    against the InSpace Option, all after the end of the TCP Payload.
    The start of the Inner Options is therefore 4 * (InOO +3) octets
    before the end of the segment, where InOO is read from within the
    InSpace Option.

    A receiver implementation will check whether Magic Number A is
    present at the end of the segment if it does not first find it at
    the start of the segment. Although the InnerOptions are located
    at the end of the TCP Payload, they are considered to be applied
    before the first octet of the TCP Payload.

SYN=0: The structure of the first non-SYN segment that contains any
    TCP Data is shown in Figure 9b).

    The receiver will find the second InSpace Option (InSpace#2)
    located SPS#1 octets from the start of the segment, where SPS#1 is
    the value of Sent Payload Size that was read from the InSpace
    Option in the previous (SYN=1) segment that started the half-
    connection. Although the Inner Options are shifted, as for the
    first segment, they are still considered to be applied at the
    start of the TCP Data in this second segment.

From the second InSpace Option onwards, the structure of the stream
reverts to that already defined in Section 2.2.1. So the value of
Sent Payload Size (SPS#2) in the second InSpace Option (InSpace #2)
defines the length of any remaining TCP Payload before the end of the
first data segment, as shown.

```
                                     TCP Data
                           .-----------------------'---------------------.
                           |               Inner Options                 |
       a) SYN=1            |          .----------'----------.            |
      +--------+----------+--------+----------+----------+---------+------+
      | BaseHdr| OuterOpts| Payload|PrefixOpts|SuffixOpts|InSpace#1|MagicA|
      +--------+----------+--------+----------+----------+---------+------+
      |        DO         |        |    SOO   |          |         |  1   |
      `------------------>|        `--------->|          |  Len    |<-----'
      |                   |        |          InOO       |<--------'      |
      |                   |        |<------------------'                  |
                          |<------------------'

       b) First SYN=0 segment in either direction
      +--------+----------+----------+---------+--------------+----------+
      | BaseHdr| OuterOpts| Payload  |InSpace#2| Inner Options| Payload  |
      +--------+----------+----------+---------+--------------+----------+
      |        DO         | SPS#1    | Len     |    InOO      | SPS#2    |
      `------------------>`--------->`-------->`------------->`--------->|
```
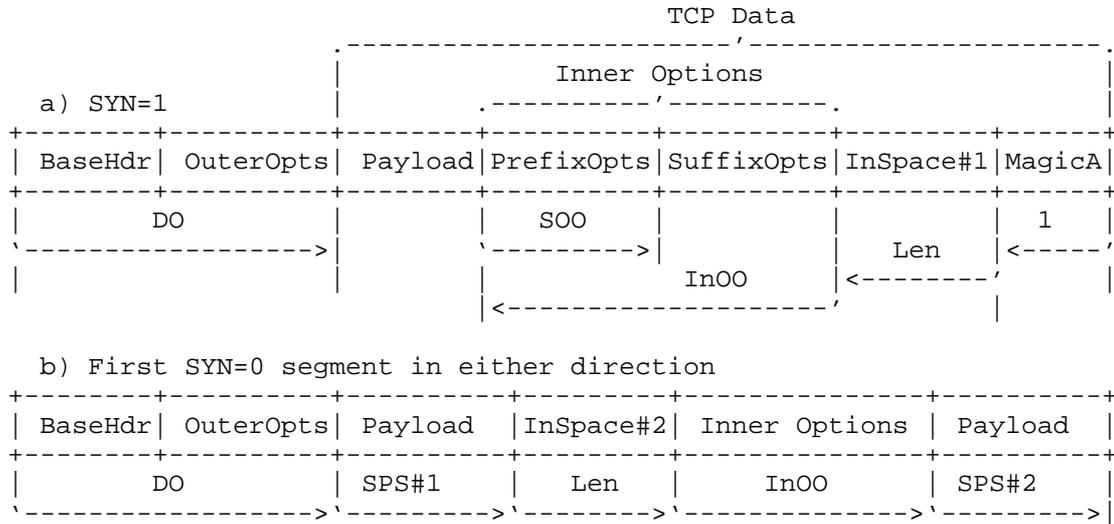
     All offsets are specified in 4-octet (32-bit) words, except SPS,
                        which is in octets.

     Figure 9: Segment Structures to Traverse DPI boxes (not to scale)

   It is recognised that having to work from the end of the first
   segment makes processing more involved. Experimental implementation
   of this approach will determine whether the extra complexity improves
   DPI box traversal sufficiently to make it worthwhile.

Appendix B. Comparison of Alternatives

B.1. Implicit vs Explicit Dual Handshake

   In the body of this specification, two variants of the dual handshake
   are defined:

   1. The implicit dual handshake (Section 2.1.1) starting with just an
       Ordinary SYN (no InSpaceO flag option) on the Ordinary
       Connection;

   2. The explicit dual handshake (Appendix A.2) starting with a SYN-O
       (InSpaceO flag option) on the Ordinary Connection.

   Both schemes double up connection state (for a round trip) on the
   Legacy Server. But only the implicit scheme doubles up connection
   state (for a round trip) on the Upgraded Server as well. On the
   other hand, the explicit scheme risks delay accessing a Legacy Server

if a middlebox discards the SYN-O (it is possible that some firewalls
will discard packets with unrecognised TCP options {ToDo: ref?}).
Table 3 summarises these points.

+--------------------------------+--------------+---------------+
| | SYN | SYN-L |
| | (Implicit) | (Explicit) |
+--------------------------------+--------------+---------------+
Minimum state on Upgraded Server	-	+
Minimum risk of delay to Legacy	+	-
Server		
+--------------------------------+--------------+---------------+

            Table 3: Comparison of Implicit vs. Explicit Dual Handshake on the
                              Ordinary Connection

There is no need for the IETF to choose between these. If the
specification allows either or both, the tradeoff can be left to
implementers at build-time, or to the application at run-time.

Initially clients might choose the Implicit Dual Handshake to
minimise delays due to middlebox interference. But later, perhaps
once more middleboxes support the scheme, clients might choose the
Explicit scheme, to minimise state on Upgraded Servers.

Appendix C. Protocol Design Issues (to be Deleted before Publication)

This appendix is informative, not normative. It records outstanding
issues with the protocol design that will need to be resolved before
publication.

Option alignment following re-segmentation: If the byte-stream is
   resegmented (e.g. by a connection splitter), the TCP options
   within the stream will not necessarily align on 4-octet word
   boundaries within the new segments.

Ossifies reliable ordered delivery into TCP design: At present it is
   theoretically possible to implement a variant of TCP that provides
   partial reliability. Inner Space as it stands would prevent a
   future partial reliable TCP, but not if out-of-order delivery were
   added, as discussed below.

Ideally Outer Options in Inner: Ideally enable Outer Options to be
   located beyond the Data Offset: i) without consuming receive
   window ii) either without consuming sequence space or, if
   otherwise, must be robust to middlebox correction; iii) delivered
   immediately on reception, not in sent order. Could use the Minion

   [Iyengar10] variant (or a similar variant) of the consistent
   overhead byte-stuffing (COBS) encoding.

Appendix D. Change Log (to be Deleted before Publication)

   A detailed version history can be accessed at
   <http://datatracker.ietf.org/doc/draft-briscoe-tcpm-inner-space/
   history/>

   From briscoe-...-inner-space-00 to briscoe-...-inner-space-01:
      Technical changes:

      * Corrected DO to 4 * DO (twice)

      * Confirmed that receive window applies to Inner Options

      * Generalised the cause of decryption/decompression from a
         previous TCP option to any previouis control message

      * Added requirement for a middlebox not to defer data on SYN

      * Latency of dual handshake is worst of two

      * Completed "Interaction with Pre-Existing TCP Implementations"
         section, covering other TCP variants, TCP in middleboxes and
         the TCP API. Shifted some TCP options to Outer only, because
         of RWND deadlock problem

      * Added two outstanding issues: i) ossifies reliable ordered
         delivery; ii) Ideally Outer in Inner.

      Editorial changes:

      * Removed section on Echo TCP option to a separate I-D that is
         mandatory to implement for inner-space, and shifted some SYN
         flood discussion in Security Considerations

      * Clarifications throughout

      * Acknowledged more review comments

   From draft-briscoe-tcpm-syn-op-sis-02 to draft-briscoe-tcpm-inner-
   space-00:
      The Inner Space protocol is a development of a proposal called the
      SynOpSis (Sister SYN options) protocol. Most of the elements of
      Inner Space were in SynOpSis, such as the implicit and explicit
      dual handshakes; the use of a magic number to flag the existence

of the option; the various header offsets; and the option
processing rules.

The main technical differences are: Inner Space extends option
space on any segment, not just the SYN; this advance requires the
introduction of the Sent Payload Size field and a general
rearrangement and simplification of the protocol format; the
option processing rules have been extended to assure compatibility
with TFO and one degree of recursion has been introduced to cater
for encryption or compression of Inner Options; The Echo option
has been added to provide a SYN-cookie-like capability. Also, the
default protocol has been pared down to the bare bones and
optional extensions relegated to appendices.

The main editorial differences are: The emphasis of the Abstract
and Introduction has expanded from a focus on just extra space
using the dual handshake to include much more comprehensive
middlebox traversal. A comprehensive Design Rationale section has
been added.

Author's Address

   Bob Briscoe
   BT
   B54/77, Adastral Park
   Martlesham Heath
   Ipswich IP5 3RE
   UK

   Phone: +44 1473 645196
   Email: bob.briscoe@bt.com
   URI: http://bobbriscoe.net/

TCP Use TLS Option
draft-rescorla-tcpinc-tls-option-01

Abstract

   This document defines a TCP option (TCP-TLS) to indicate that TLS
   should be negotiated on a given TCP connection.

Status of This Memo

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

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

   Internet-Drafts are draft documents valid for a maximum of six months
   and may be updated, replaced, or obsoleted by other documents at any
   time. It is inappropriate to use Internet-Drafts as reference
   material or to cite them other than as "work in progress."

   This Internet-Draft will expire on April 30, 2015.

Table of Contents

1. Introduction

   RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH The source for this
   draft is maintained in GitHub. Suggested changes should be submitted
   as pull requests at https://github.com/ekr/tcpinc-tls. Instructions
   are on that page as well.

   The TCPINC WG is chartered to define protocols to provide ubiquitous,
   transparent security for TCP connections.

   While TLS [RFC5246] is by far the most popular mechanism for securing
   TCP data, adding it to a given protocol requires some sort of
   coordination; if a client just tries to initiate TLS with a non-TLS
   server, the server will most likely reject the protocol messages
   because they do not conform to its expectations for the application
   layer protocol. This coordination can take a number of forms,
   including:

   o An external signal in the URL that the client should do TLS (e.g.,
      "https:")

   o Using a separate port for the secure and non-secure versions of
      the protocol.

   o An extension to the application protocol to negotiate use or non-
      use of TLS ("STARTTLS")

   While mechanisms of this type are in wide use, they all require
   modifications to the application layer and thus do not meet the goals
   of TCPINC. This document describes a TCP option which allows a pair
   of communicating TCP endpoints to negotiate TLS use automatically

without modifying the application layer protocols, thus allowing for
transparent deployment.

2. Overview

The basic idea behind the TCP-TLS option is simple. The SYN and SYN/
ACK messages carry TCP options indicating the willingness to do TLS
and some basic information about the expected TLS modes. If both
sides want to do TLS and have compatible modes, then the application
data is automatically TLS protected prior to being sent over TCP.
Otherwise, the application data is sent as usual.

```
        Client                                  Server

        SYN + TCP-TLS ->
                                    <- SYN/ACK + TCP/TLS
        ACK ->
        <--------------- TLS Handshake --------------->
        <--------- Application Data over TLS ---------->
```
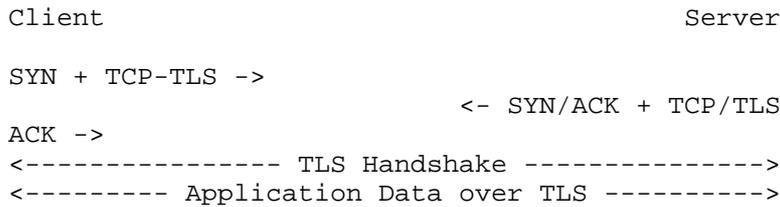
            Figure 1: Negotiating TLS with TCP-TLS

```
        Client                                  Server

        SYN + TCP-TLS ->
                                    <- SYN/ACK
        ACK ->
        <--------- Application Data over TLS ---------->
```
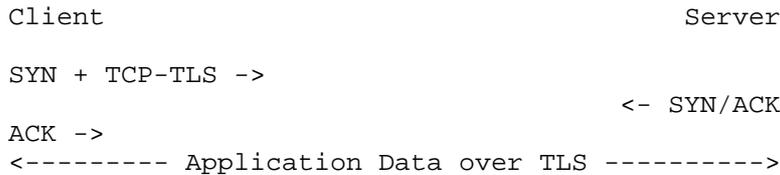
                Figure 2: Fall back to TCP

If use of TLS is negotiated, the data sent over TCP simply is TLS
data in compliance with {{RFC5246}.

3. Extension Definition

The TCP-TLS option is very simple. For the normal case where each
side knows who is the passive and who is the active opener, the
option is empty. I.e.

```
        +------------+------------+
        |  Kind=XX   | Length = 2 |
        +------------+------------+
```
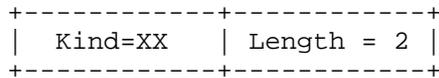
In this case, the active opener MUST take on the role of TLS Client.

In the abnormal case of simultaneous open, the option includes a tiebreaker value.

```
+------------+-----------+-----------+------------+
|  Kind=XX   | Length = 8 |        Tiebreaker     |
+------------+-----------+-----------+------------+
|                    Tiebreaker                   |
+-------------------------------------------------+
```

The tiebreaker field is a 48-bit value which is used to determine the TLS roles, with the highest value being the TLS client and the lowest value being the TLS server. Applications MUST generate the tiebreaker randomly. If both sides generate the same tiebreaker value, then TCP-TLS MUST NOT be used (this has a vanishing probability of happening by accident.)

The default mode of operation MUST be the ordinary client/server mode and implementations MUST only use the simultaneous open mode if instructed by an application. If an implementation in simultaneous open mode receives an option without a tiebreaker, it MUST treat that tiebreaker as 0. If simultaneous open mode is not in use, and implementations detect a simultaneous open, then they MUST NOT try to negotiate TLS, regardless of the presence of this option.

If an endpoint sends the TCP-TLS option and correctly receives it from the other side it SHALL immediately negotiate TLS, taking on the role described above.

Once the TLS handshake has completed, all application data SHALL be sent over that negotiated TLS channel. Application data MUST NOT be sent prior to the TLS handshake.

If the TLS handshake fails for non-cryptographic reasons such as failure to negotiate a compatible cipher or the like, endpoints SHOULD behave as if the the TCP-TLS option was not present. This is obviously not the conventional behavior for TLS failure, but as the entire idea here is to be opportunistic and the attacker can simply suppress the TCP-TLS option entirely, this provides the maximum robustness against broken intermediaries. If the TLS handshake fails for cryptographic reasons that indicate damage to the datastream (e.g., a decryption failure or a Finished failure) then the endpoints SHOULD signal a connection failure, as this suggests that there is a middlebox modifying the data and there is a reasonable chance that the state is now corrupted.

4. Transport Integrity

   The basic operational mode defined by TCP-TLS protects only the
   application layer content, but not the TCP segment metadata. Upon
   receiving a packet, implementations MUST first check the TCP checksum
   and discard corrupt packets without presenting them to TLS. If the
   TCP checksum passes but TLS integrity fails, the connection MUST be
   torn down.

   Thus, TCP-TLS provides automatic security for the content, but not
   protection against DoS-style attacks. For instance, attackers will
   be able to inject RST packets, bogus application segments, etc.,
   regardless of whether TLS authentication is used. Because the
   application data is TLS protected, this will not result in the
   application receiving bogus data, but it will constitute a DoS on the
   connection.

   This attack can be countered by using TCP-TLS in combination with
   TCP-AO [RFC5925], as follows:

   1. The TLS connection is negotiated using the "tcpao" ALPN
       [I-D.ietf-tls-applayerprotoneg] indicator.

   2. Upon TLS handshake completion, a TLS Exporter [RFC5705] is used
       to generate keying material of appropriate length using exporter
       label TBD.

   3. Further packets are protected using TCP-AO with the generated
       keys.

   The Finished messages MUST NOT be protected with AO. The first
   application data afterwards MUST be protected with AO. Note that
   because of retransmission, non-AO packets may be received after AO
   has been engaged; they MUST be ignored.

   [[OPEN ISSUE: How do we negotiate the parameters? Do we need a
   use_ao option like with RFC 5764? Is ALPN really what we want
   here?]]

   [[TODO: verify that the state machine matches up here.]]

5. Implementation Options

   There are two primary implementation options for TCP-TLS:

   o Implement all of TCP-TLS in the operating system kernel.

   o Implement just the TCP-TLS negotiation option in the operating
      system kernel with an interface to tell the application that TCP-
      TLS has been negotiated and therefore that the application must
      negotiate TLS.

   The former option obviously achieves easier deployment for
   applications, which don't have to do anything, but is more effort for
   kernel developers and requires a wider interface to the kernel to
   configure the TLS stack. The latter option is inherently more
   flexible but does not provide as immediate transparent deployment.
   It is also possible for systems to offer both options.

6. TLS Profile

   Implementations of this specification MUST at minimum support TLS 1.2
   [RFC5246] and MUST support cipher suite XXX. Implementations MUST
   NOT negotiate versions of TLS prior to TLS 1.2. [[OPEN ISSUE: What
   cipher suites? Presumably we require one authenticated and one
   anonymous cipher suite, all with GCM.]] [[OPEN ISSUE: If TLS 1.3 is
   ready, we may want to require that.]]

7. Channel Bindings

   This specification is compatible with external authentication via TLS
   Channel Bindings [RFC5929]. If Channel Bindings are to be used, the
   TLS Extended Master Secret Extension [I-D.ietf-tls-session-hash]

8. NAT/Firewall considerations

   If use of TLS is negotiated, the data sent over TCP simply is TLS
   data in compliance with {{RFC5246}}. Thus it is extremely likely to
   pass through NATs, firewalls, etc. The only kind of middlebox that
   is likely to cause a problem is one which does protocol enforcement
   that blocks TLS on arbitrary (non-443) ports but _also_ passes
   unknown TCP options. Although no doubt such devices do exist,
   because this is a common scenario, a client machine should be able to
   probe to determine if it is behind such a device relatively readily.

9. IANA Considerations

   IANA [shall register/has registered] the TCP option XX for TCP-TLS.

   IANA [shall register/has registered] the ALPN code point "tcpao" to
   indicate the use of TCP-TLS with TCP-AO.

10. Security Considerations

   The mechanisms in this document are inherently vulnerable to active
   attack because an attacker can remove the TCP-TLS option, thus
   downgrading you to ordinary TCP. Even when TCP-AO is used, all that
   is being provided is continuity of authentication from the initial
   handshake. If some sort of external authentication mechanism was
   provided or certificates are used, then you might get some protection
   against active attack.

   Once the TCP-TLS option has been negotiated, then the connection is
   resistant to active data injection attacks. If TCP-AO is not used,
   then injected packets appear as bogus data at the TLS layer and will
   result in MAC errors followed by a fatal alert. The result is that
   while data integrity is provided, the connection is not resistant to
   DoS attacks intended to terminate it.

   If TCP-AO is used, then any bogus packets injected by an attacker
   will be rejected by the TCP-AO integrity check and therefore will
   never reach the TLS layer. Thus, in this case, the connection is
   also resistant to DoS attacks, provided that endpoints require
   integrity protection for RST packets. If endpoints accept
   unauthenticated RST, then no DoS protection is provided.

11. References

11.1. Normative References

   [I-D.ietf-tls-applayerprotoneg]
             Friedl, S., Popov, A., Langley, A., and S. Emile,
             "Transport Layer Security (TLS) Application Layer Protocol
             Negotiation Extension", draft-ietf-tls-applayerprotoneg-05
             (work in progress), March 2014.

   [I-D.ietf-tls-session-hash]
             Bhargavan, K., Delignat-Lavaud, A., Pironti, A., Langley,
             A., and M. Ray, "Transport Layer Security (TLS) Session
             Hash and Extended Master Secret Extension", draft-ietf-
             tls-session-hash-02 (work in progress), October 2014.

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

   [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security
             (TLS) Protocol Version 1.2", RFC 5246, August 2008.

   [RFC5705] Rescorla, E., "Keying Material Exporters for Transport
             Layer Security (TLS)", RFC 5705, March 2010.

   [RFC5925] Touch, J., Mankin, A., and R. Bonica, "The TCP
              Authentication Option", RFC 5925, June 2010.

11.2. Informative References

   [I-D.bittau-tcp-crypt]
              Bittau, A., Boneh, D., Hamburg, M., Handley, M., Mazieres,
              D., and Q. Slack, "Cryptographic protection of TCP Streams
              (tcpcrypt)", draft-bittau-tcp-crypt-04 (work in progress),
              February 2014.

   [RFC5929] Altman, J., Williams, N., and L. Zhu, "Channel Bindings
              for TLS", RFC 5929, July 2010.

Author's Address

   Eric Rescorla
   Mozilla

   EMail: ekr@rtfm.com

                         A DTLS Extension for TCP
                       draft-thomson-tcpinc-dtls-00

Abstract

   Opportunistic security is provided for TCP using a modified DTLS.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

   TCP [RFC0793] is a widely used protocol.

   As part of a general "secure all the things" effort, the IETF is
   defining opportunistic security options for all the protocols it
   maintains. Opportunistic security ensures that we accelerate the
   eventual heat death of the universe, and discourages certain classes
   of attack [RFC7258].

   Opportunistic approaches are the most practical way to ensure wider
   deployment of security because they don't immediately depend on
   solving hard problems like authentication.

In that spirit, reusing existing security protocols reduces the cost
to implement, deploy and analyse new protocol modifications. TLS
[RFC5246] and DTLS [RFC6347] represent the current best in class
security protocols.

This specification defines how DTLS can be used to protect TCP. This
addresses the requirements outlined in [I-D.bellovin-tcpsec]. A
small modification to the TCP record layer allows for the protection
of the TCP pseudo header, with an allowance for NAPT (editor: why
does Bellovin even suggest that protection of IP/port is even
feasible?) and per-option opt-out.

In addition, all the features of DTLS are made available:

o Cipher suite negotiation and agility

o Years of security analysis

o Downgrade protection in the handshake

o Session bindings

o Optional peer authentication

o A range of available extensions

In addition to this, new upgrades to DTLS can be trivially added.
Thus, improvements to algorithms or the DTLS handshake are entirely
portable.

1.1. Terminology

The usual. [RFC2119] explains what those are.

2. DTLS Layering

This extension to TCP places a continuous sequence of DTLS records as
the payload of TCP. These records provide confidentiality and
integrity protection for their content, plus integrity protection for
the TCP header and pseudoheader.

An option negotiates the use of this extension. This option is added
to the SYN message to indicate support, and to the ACK message to
indicate acceptance.

Once enabled, all DTLS records, including handshake messages, are
carried as TCP data. The data for the protected TCP stream is the
concatenated content of DTLS messages.

TCP clients are automatically entered into the DTLS client role; and
TCP servers automatically enter the DTLS server role. Where TCP
simultaneous open is used, a lottery determines the roles Section 5.

3. DTLS Record Protection Option

   This option is used to negotiate the use of DTLS. It is assigned a
   TCP option kind of 0xTBD Section 8.

   The format of the DTLS record protection option is a single octet
   flags field, followed by a list of protected option kinds.

3.1. DTLS Record Protection Flags

   The content of the flags field is a bit pattern of features. The
   following features are defined in this document:

   o FORBID_NAPT: Bit 0 (the first and most significant bit of the
      first octet) being set indicates that DTLS protection is to be
      extended to addressing elements, see Section 4.3.

   A client can set these bits to request the defined alterations to the
   protocol. A server can accept these alterations by including these
   in its ACK message, or it can reject the alterations by clearing the
   bit.

   All bits in this option MUST be set to zero unless they are
   explicitly understood. A sender MUST remove trailing octets that
   have all zero values from the option.

   An IANA registry is established to maintain these bits Section 8.2.

3.2. Protection Option Kinds

   The DTLS record protection option includes a list of the TCP options
   that are covered by DTLS integrity protection, each occupying a
   single octet. Just as TCP options are terminated by a zero octet,
   this list is terminated by a zero value.

   Any data following this list is reserved for extension and MUST be
   ignored.

4. Modified DTLS AEAD Operation

   This mechanism MUST be used with an Authenticated Encryption with
   Additional Data (AEAD) mode. The DTLS record layer is modified to
   provide integrity protection for the TCP pseudoheader and header by
   including this as part of the additional data.

   An important characteristic of this is that records are protected as
   though each individual DTLS record is part of a unique TCP segment.
   This ensures that repacketization by middleboxes does not result in
   records being marked as invalid.

   TCP middleboxes can, and sometimes do, split or coalesce TCP
   segments. This affects the calculation of the authenticated data
   that is input to the AEAD protection.

   To prevent this from invalidating integrity checks unnecessarily, the
   associated data passed to the AEAD algorithm contains a modified
   value of the TCP header and pseudoheader.

   For a sender that transmits a single DTLS record in each TCP segment
   with only protected TCP options, this demands no additional
   calculation. However, a receiver needs to construct the TCP header
   and pseudoheader. The length of this packet is based on the length
   of the DTLS record, with the value of protected TCP options being
   extracted from the TCP header of the segment that carries the first
   byte of the DTLS record.

   In TLS and DTLS, the additional data that is protected by the AEAD
   function is [RFC5246]:

   additional_data = seq_num + TLSCompressed.type +
                     TLSCompressed.version + TLSCompressed.length;

   where "+" denotes concatenation.

   This specification expands the fields that are protected to include a
   constructed TCP pseudoheader and header as follows:

   tcp_additional_data = pseudoheader + tcp_header +
                         additional_data;

   Construction of the "pseudoheader" and "tcp_header" portions of the
   authenticated data are described in the following sections.

4.1. TCP Pseudoheader Construction

   The pseudoheader that is used for AEAD input depends on the IP
   version in use, for IPv4 [RFC0793], with length of fields in bits
   shown in parentheses:

   pseudoheader_v4 = source_address(32) + destination_address(32) +
                     zero(8) + protocol(8) + tcp_length(16)

   Or for IPv6 [RFC2460]:

```
pseudoheader_v6 = source_address(128) + destination_address(128) +
                  tcp_length(32) + zero(24) + protocol(8)
```

In both cases, the value for "tcp_length" is derived by constructing a TCP header as described in Section 4.2.

The values for "source_address" and "destination_address" are replaced with zero bits, unless the FORBID_NAPT flag is enabled. Setting these values to zero permits the use of NAPT devices.

4.2. TCP Header Construction

In order to ensure that the protocol is robust in the presence of middleboxes, unprotected TCP options are removed from the TCP header before applying protection.

```
tcp_header = source_port(16) + destination_port(16) +
             sequence_number(32) + acknowledgement_number(32) +
             data_offset(4) + flags(12) + window(16) +
             checksum(16) + urgent_pointer(16) + options(?)
```

The following construction rules apply:

source_port and destination_port: These fields MUST be replaced with zero bits unless the FORBID_NAPT flag is enabled for the session. Setting these values to zero permits the use of port translation.

sequence_number: This field MUST be set to the sequence number corresponding to the first octet of the DTLS record. If multiple segments are combined into a single packet, this will be different to the sequence number that appears in the TCP header.

acknowledgement_number and window: These fields MUST be replaced with zero bits. Removing the acknowledgement and congestion window from integrity protection does provide some opportunities to an on-path attacker Section 7.2.

data_offset: The data offset MUST be set to the size of the modified TCP header.

flags: The reserved and flags part of the TCP header is protected.

checksum: This field MUST be replaced with zero bits, just as it is when the TCP checksum is calculated.

urgent_pointer: The urgent pointer is protected.

   options: The set of options that are included under protection are
      included. Options that are not protected are removed.
      Section 3.2 described how options are selected for protection.
      The list of options is terminated with an option of kind 0x0 and
      padding to a multiple of 32 bits with zero octets.

   This construction permits the addition and removal of options by
   middleboxes, as long as they are not in the list of options that are
   protected. It also permits repacketization and acknowledgment.

4.3. Forbid NAPT

   The DTLS record protection option Section 3 contains a FORBID_NAPT
   bit that can be used to signal that network address and port
   translation (NAPT) is forbidden.

   If the FORBID_NAPT option is not set, addressing information is
   replaced with zero values. This is the IP (v4 or v6) address fields
   in the pseudoheader, and the source and destination port numbers.

   Why anyone in their right mind would do this is beyond me, but it's
   in the requirements and this would seem to be sufficient to address
   those, albeit by making the whole mechanism more complex.

5. DTLS Role Selection

   Ordinarily, the role of DTLS client is assumed by the peer that sends
   the first TCP SYN packet (the TCP client), and the role of DTLS
   server is assumed by the peer that responds (the TCP server).

   Peers that perform a TCP simultaneous open - that is, where both
   peers simultaneously send SYN packets to open a connection, often to
   work around middlebox limitations - are assigned client and server
   roles in DTLS based on the following rules.

   If only one peer provides a DTLS handshake in TCP fast open data
   [I-D.ietf-tcpm-fastopen], then that peer becomes the client. Note
   that including the DTLS handshake message in the initial SYN packet
   is only safe if there is a previous confirmation from a server that
   it supports this protocol (see Section 6.1).

   If neither or both peers provide the DTLS handshake option, then the
   peer that selects the numerically highest value for their
   ClientRandom assumes the client role. In the absence of the DTLS
   handshake option, role allocations are not determined until a
   ClientHello message is exchanged.

6. Design Characteristics

   This section outlines a number of considerations that allow this
   protocol to actually be implemented.

6.1. Interaction with TCP Fast Open

   TCP fast open [I-D.ietf-tcpm-fastopen] can be used to mitigate the
   additional latency cost imposed by the DTLS handshake. However, this
   represents a risk, since the payload of the initial packet is
   directly passed to an application if the opportunistic security
   option is not negotiated.

   Adding data to an initial SYN is therefore only possible if there is
   a previous indication that a server supports the combination of TCP
   fast open and opportunistic security in combination.

   A server that provides a TCP fast open cookie for an encrypted
   connection MUST accept encryption on future connections with that
   cookie, or reject the connection. This ensures that clients are able
   to send a DTLS handshake message in the initial SYN packet.

6.2. Zero Length DTLS Data

   [RFC5246], Section 6.2 notes that the TLS record layer protects non-
   zero length blocks. This use of DTLS requires that frames be
   permitted to be empty, relying solely on integrity protection of the
   associated data.

      This does not mean that the TCP segment contains no data, since it
      will contain the DTLS record header (including the explicit nonce,
      if any, and any bits produced by the AEAD cipher to ensure
      integrity).

6.3. Unauthenticated Acknowledgments

   TCP segments that only acknowledge receipt of data, or update the
   receive window do not require authentication, since the corresponding
   fields are not protected. These frames can be accepted and
   processed, as long as only the receive window is updated.

   By the same logic, protection of the TCP window scaling option
   [RFC1323] and the selective acknowledgment (SACK) option [RFC2018]
   are not made mandatory. These SHOULD NOT be added to the list of
   protected options Section 3.2.

6.4. Interaction with DTLS Replay Protection

   TCP segment retransmission and reassembly requires that a sender be
   able to retransmit. These frames will be retransmitted with the same
   data, including the DTLS serial number. To avoid having
   retransmissions erroneously discarded, any DTLS replay protection
   needs to allow for replay of records that appear in unacknowledged
   segments.

6.5. TCP Keep-Alive

   This protocol does not protect TCP keep-alive segments [RFC1122];
   that is, segments that are sent purely to ensure that the connection
   is maintained through middleboxes. These can contain a single junk
   byte from just prior to the start of the congestion window. These
   segments are discarded without being validated.

   This differs from [I-D.bittau-tcp-crypt], which protects keep-alive
   segments. Protection ensures that an attacker is unable to prolong
   the lifetime of a connection that is otherwise unwanted.

   Since an unwanted connection can be terminated with an authenticated
   segment that bears a FIN or RST bit, this concern is unwarranted.

6.6. Unprotected RST Segments

   Existing TCP implementations, particularly middleboxes, rely TCP RST
   to terminate connections. If RST authentication is required, then it
   becomes impossible for a node which is not part of the association
   (either because it is a middlebox or because it is a legitimate
   endpoint which has lost state) to terminate the connection. An
   implementation MAY choose to respect an unauthenticated RST to permit
   these uses.

   (Note: we may want to provide an option that the middlebox can
   include in a RST to prove that it is on-path to make this a little
   easier to accept.)

6.7. Cipher Suite Selection

   Implementations MUST support the TLS_BLAH_WITH_BLAH_BLAH cipher
   suite.

   Implementations MUST NOT offer non-AEAD modes and MUST terminate the
   connection if a non-AEAD mode if one is erroneously offered.

7. Security Considerations

   None of this document mandates any level of authentication for peers,
   which opens up all sorts of active attacks.

7.1. NAPT

   The choice to protect a TCP connection from addressing modification
   prevents network address and port translation from altering the
   addressing information on a connection. Unfortunately, this is a
   procedure that much of the Internet relies on. Enabling this feature
   is likely to break a lot of uses, but failure to use it exposes the
   connection to trivial re-routing attacks.

   In the absence of peer authentication, and where there is a high
   level of assurance that no NAPT is being used for a communications
   path, this protection might be used. Of course, any protection this
   provides is trivially circumvented by an on-path attacker.

7.2. Acknowledgments and Congestion Window Protection

   This design permits a middlebox to generate acknowledgments and to
   perform repacketization. This opens a number of denial of service
   avenues for malicious middleboxes. Falsifying window advertisements
   can cause a sender to send more packets than might otherwise be sent.
   Similarly, sending a reduced acknowledgment sequence number can cause
   excessive retransmission. In a similar fashion, retransmissions can
   be suppressed by sending inflated acknowledgment sequence numbers.

   These are options that are already available to an on-path attacker.

7.3. Traffic Redirection

   Without the FORBID_NAPT flag enabled, it's possible for a middlebox
   to rewrite addressing information so that this flow. If only
   authenticated RST and FIN segments are accepted by the TCP stack, the
   target of this flow - who doesn't have access to the traffic keys -
   is unable to do anything to end the flow of data.

   This isn't particularly interesting as an attack, since we have to
   assume that any middlebox capable of this is also capable of just
   generating the same volume of packets toward the victim.

7.4. Peer Authentication

   In order to have this deployed, peers will have to avoid relying on
   authentication. That means that this is open to active attacks.

Implementations might consider using some form of key continuity. Clients SHOULD avoid key continuity for different servers to avoid tracking by correlating keying material. Full continuity might be more applicable for servers, where key continuity does not create any special tracking ability.

(This probably needs work.)

8. IANA Considerations

This document registers a new TCP option kind, and establishes a registry to maintain its contents.

8.1. Registration of DTLS Record Protection Option Kind

This document registers the DTLS record protection option with a TCP option kind of 0xTBD.

The format of this option is described in Section 3

8.2. Registry for DTLS Record Protection Flags

IANA will maintain a registry of "TCP DTLS Record Protection Flags" under the "Service Names and Transport Protocol Port Numbers" group of registries.

This registry controls a contiguous space starting from bit 0 to 2023 (inclusive). New registrations in this registry require IETF review [RFC5226], with the following information:

Bit Number: The bit number being assigned

Purpose: A brief description of the feature.

Specification: A reference to the specification that defines the feature.

The initial contents of this registry are:

Bit Number: 0

Purpose: Enables protection of addressing information.

Specification: This document.

9. References

9.1. 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.

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

   [RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6
              (IPv6) Specification", RFC 2460, December 1998.

   [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an
              IANA Considerations Section in RFCs", BCP 26, RFC 5226,
              May 2008.

   [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security
              (TLS) Protocol Version 1.2", RFC 5246, August 2008.

   [RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer
              Security Version 1.2", RFC 6347, January 2012.

9.2. Informative References

   [I-D.bellovin-tcpsec]
              Bellovin, S., "Problem Statement and Requirements for a
              TCP Authentication Option", draft-bellovin-tcpsec-01 (work
              in progress), July 2007.

   [I-D.bittau-tcp-crypt]
              Bittau, A., Boneh, D., Hamburg, M., Handley, M., Mazieres,
              D., and Q. Slack, "Cryptographic protection of TCP Streams
              (tcpcrypt)", draft-bittau-tcp-crypt-04 (work in progress),
              February 2014.

   [I-D.ietf-tcpm-fastopen]
              Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP
              Fast Open", draft-ietf-tcpm-fastopen-09 (work in
              progress), July 2014.

   [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.

   [RFC7258] Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an
              Attack", BCP 188, RFC 7258, May 2014.

Author's Address

   Martin Thomson
   Mozilla

   Email: martin.thomson@gmail.com

          A TCP Authentication Option Extension for Payload Encryption
                      draft-touch-tcp-ao-encrypt-01.txt

Status of this Memo

Copyright Notice

Section 4.e of the Trust Legal Provisions and are provided without
warranty as described in the Simplified BSD License.

Abstract

This document describes an extension to the TCP Authentication
Option (TCP-AO) to encrypt the TCP segment payload in addition to
providing TCP-AO's authentication of the payload, TCP header, and IP
pseudoheader. This extension augments how the packet contents and
headers are processed and which keys are derived, and adds a
capability for in-band coordination of unauthenticated Diffie-
Hellman key exchange at connection establishment. The extension
preserves key rollover coordination and protection of long-lived
connections.

Table of Contents

1. Introduction

This document describes an extension to the TCP Authentication
Option (TCP-AO) [RFC5925] called TCP-AO-ENC to support its use to
encrypt TCP segment payload contents in addition to authenticating
the segment. TCP-AO-ENC is intended for use where TCP user data
privacy is required and where TCP control protocol protection is
also needed.

TCP-AO-ENC supports two different modes: authenticated encryption
and unauthenticated (BTNS-style) encryption [RFC5387]. Authenticated
mode (ENC-AUTH) relies on out-of-band coordination of master key
tuples in the same way as TCP-AO, and protects all segments of a
connection. Unauthenticated (ENC-BTNS) mode supports in-band
unsigned Diffie-Hellman key exchange during the initial SYN, and
protects connections from all except man-in-the-middle attacks
during connection establishment.

This document assumes detailed familiarity with TCP-AO [RFC5925].
TCP-AO-ENC extends how TCP-AO generates traffic keys and how those
keys are used to process TCP segment headers and payloads, but does
not otherwise alter other aspects of the TCP-AO mechanism [RFC5926].

2. Conventions used in this document

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].
When used in lower case, these words have their conventional meaning
and do not convey the interpretations in RFC-2119.

3. Background

The premise of TCP-AO-ENC is that it might be useful to allow TCP-AO
to encrypt TCP segment payloads, in addition to authenticating the
entire segment.

This is accomplished by the following additions, as a preview:

o An encryption flag to indicate when segment payload encryption is
   used.

o Traffic encryption key, in addition to the TCP-AO traffic
   (authentication) key. TCP-AO-ENC can be used with only symmetric
   ciphers that avoiding the need for padding (stream ciphers).

o Augment input and output processing to include
   encryption/decryption.

TCP-AO-ENC does not change any other aspects of TCP-AO [RFC5925],
and is compatible with TCP-AO-NAT [RFC6978]. TCP-AO-NAT is intended
for use only where coordinated between endpoints for connections
that match the shared Master Key Tuple (MKT) parameters, as with all
other MKT parameters.

4. Authenticated and Unauthenticated Modes

   TCP-AO-ENC includes two modes: authenticated (ENC-AUTH) and
   unauthenticated (ENC-BTNS). The latter is consistent with the
   "Better Than Nothing Security" approach to protect communication in
   the absence of public key infrastructure (PKI) or pre-shared keys
   [RFC5387].

   ENC-AUTH mode operates in the same way as original TCP-AO
   authentication, by relying on out-of-band Master Key Tuples (MKTs)
   that are deployed in advance of new connections. All segments of
   connections covered by ENC-AUTH are encrypted and authenticated
   using keying material derived from those MKTs.

   ENC-BTNS mode can be used without an out-of-band key exchange
   protocol. It exchanges unauthenticated, unsigned Diffie-Hellman
   nonces during connection establishment in-band, and uses that
   information to derive keys used to protect the remainder of the
   connection. ENC-BTNS mode is susceptible to man-in-the-middle
   attacks in which the adversary both participates in the initial
   nonce exchange and processes subsequent segments; this protection
   increases the effort of the attacker and can help avoid low-effort
   DDOS attacks that disrupt established connections [RFC4953].

   Because ENC-BTNS uses in-band nonce exchange only during the initial
   SYN, TCP-AO key rollover is not used in that mode. The KeyIDs used
   during the nonce exchange are recorded and used throughout the
   connection.

5. Extension for Payload Encryption

   The following describe the additions to TCP-AO needed to support
   TCP-AO-ENC.

5.1. Additional Master Key Tuple components

   TCP-AO-ENC augments the MKT as follows; as with other MKT
   components, these MUST NOT change during a connection:

   o TCP encryption mode. This indicates the use and mode (ENC-AUTH
      or ENC-BTNS) of segment payload encryption, or is clear when
      encryption is not used (e.g., for conventional TCP-AO).

   o Encryption Key Derivation Function (E-KDF). Indicates the key
      derivation function and its parameters, as used to generate
      traffic encryption keys from master keys in the same way that the
      TCP-AO KDG generates traffic (authentication) keys.

o Encryption algorithm. Indicates the encryption algorithm and its
   parameters as used for encrypted connections.

PTCP-AO-ENC processes TCP packets in the same way as TCP-AO, except
that it replaces the authentication input and output processing as
follows:

5.2. Additional traffic keys

TCP-AO-ENC uses the E-KDF to derive four additional keys used for
traffic encryption:

o Send_SYN_traffic_encryption_key

o Receive_SYN_traffic_encryption_key

o Send_other_traffic_encryption_key

o Receive_other_traffic_encryption_key

5.3. Per-Connection TCP-AO Parameters

The per-connection TCP-AO parameters are not affected by the use of
TCP-AO-ENC-AUTH, except that MKTs indicated by Current_key and
Rnext_key would indicate the use of payload encryption.

The per-connection TCP-AO parameters for TCP-AO-ENC-BTNS are
augmented by the addition of the following Diffie-Hellman nonces:

o Send_nonce. The locally-generated Diffie-Hellman nonce.

o Receive_nonce. The Diffie-Hellman nonce generated by the remote
   end of the connection.

These nonces are exchanged during the initial SYN exchange in ENC-
BTNS mode; for ENC-AUTH mode, similar information is exchanged out-
of-band and is used to derive the encryption keys. KeyIDs used with
these nonces are recorded during nonce exchange and used for the
remainder of the connection.

The use of payload encryption as specified in these MKTs SHOULD NOT
change during a TCP connection.

5.4. Traffic Encryption Key Derivation Functions

   Traffic encryption keys are derived from the MKTs using the E-KDF,
   in the same way and used on the same segments as their corresponding
   authentication keys, e.g.:

   o Send_SYN_traffic_encryption_key / Send_SYN_traffic_key

   o Receive_SYN_traffic_encryption_key / Receive_SYN_traffic__key

   o Send_other_traffic_encryption_key / Send_other_traffic__key

   o Receive_other_traffic_encryption_key / Receive_other_traffic_key

6. TCP-AO-ENC Interaction with TCP

   TCP-AO-ENC augments TCP segment send and receive processing to
   include encryption/decryption. Note that the encryption
   initialization vector MAY depend on TCP header state, but MUST NOT
   depend on the processing of previous segments because segments may
   arrive (and need to be decrypted) out of order.

6.1. Sending TCP Segments

   For ENC-BTNS, initial SYN and SYN-ACK are used to establish the
   Diffie-Hellman nonces as follows:

   o Initial SYN and SYN-ACK. The initial SYN (SYN and not ACK) and
      SYN-ACK segments are not encrypted or authenticated. Instead,
      their HMAC field contains the 128-bit DIffie-Hellman nonce in
      network-standard byte order.

   Because these segments are not authenticated or encrypted, they
   SHOULD NOT contain user data. In a typical client-server system,
   user data usually commences in other segments anyway.

   All other TCP segments are processed as follows:

   1. The segment payload is encrypted in-place using the traffic
      encryption key.

   2. The segment is authenticated using TCP-AO as per [RFC5925].

6.2. Receiving TCP Segments

   For ENC-BTNS, initial SYN and SYN-ACK are used to establish the
   Diffie-Hellman nonces as follows:

   o Initial SYN and SYN-ACK. The 128-bit unauthenticated Diffie-
      Hellman nonce is extracted from the HMAC field, and used to
      construct the encryption and traffic keys for the connection.
      Because these segments are not encrypted or authenticated, no
      further processing is required.

   All other incoming TCP segments are processed as follows:

   1. TCP-AO authenticates the segment, including discarding it if
      authentication fails, as per [RFC5925].

   2. The segment payload is decrypted in-place using the traffic
      encryption key.

6.3. Other TCP Impact

   TCP-AO-ENC has no impact on TCP beyond that of TCP-AO, including
   impact on TCP header size, connectionless resets, and ICMP handling.

   TCP-AO-ENC is compatible with the use of TCP-AO-NAT if traversal of
   NAT boxes is desired.

7. Security Considerations

   TCP-AO-ENC augments TCP-AO to provide segment payload privacy.

   TCP-AO-ENC relies on TCP-AO's authentication to avoid replay attacks
   and to ensure that the segments originate from the intended source.

   TCP-AO-ENC supports only stream ciphers because the TCP segment must
   be encrypted and decrypted in-situ. Support for padding would
   require additional option space to indicate the original message
   length, and this complication does not seem necessary.

   The design of TCP-AO-ENC can support either symmetric or asymmetric
   keys. However, because TCP-AO derives traffic (authentication) keys
   from MKTs using KDFs, it was deemed sufficient that TCP-AO-ENC
   derive traffic encryption keys from MKTs using E-KDFs in a similar
   manner, and both endpoints would thus derive the same traffic
   encryption keys just as they derive the same traffic
   (authentication) keys. Extensions of TCP-AO-ENC to support
   asymmetric keying are possible if traffic keys are managed using an
   out-of-band mechanism, but not if they are derived from MKTs.

   ENC-AUTH has no additional security considerations. ENC-BTNS cannot
   authenticate or encrypt the segments used for nonce exchange, i.e.,
   the initial SYN and SYN-ACK. As a result, ENC-BTNS is susceptible to

man-in-the-middle attacks during connection establishment, but
remains useful to ensure that established connections are protected.

8. To be completed...

Where are required algorithms specified? This doc or a separate one?

o E-KDF - also, can a MKT use the same alg for KDF and E-KDF?

o Encryption algorithm - possibilities include AES CTR (CTR initial
   value can be the ESN) or AES CBC and Camellia CBC as per TLS 1.2.

9. IANA Considerations

(TO BE CONFIRMED, BUT FOR NOW:)

There are no IANA considerations for this document. This section can
be removed upon publication as an RFC.

10. References

10.1. Normative References

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

   [RFC5925] Touch, J., A. Mankin, R. Bonica, "The TCP Authentication
             Option", RFC 5925, Jun. 2010.

   [RFC5926] Lebovitz, G., E. Rescorla, "Cryptographic Algorithms for
             the TCP Authentication Option (TCP-AO)", RFC 5926, June
             2010.

10.2. Informative References

   [RFC4953] J. Touch, "Defending TCP Against Spoofing Attacks," RFC
             4953, July 2007.

   [RFC5387] Touch, J., Black, D., and Y. Wang, "Problem and
             Applicability Statement for Better-Than-Nothing Security
             (BTNS)", RFC 5387, November 2008.

   [RFC6978] Touch, J., "A TCP Authentication Option Extension for NAT
             Traversal", RFC 6978, July 2013.

11. Acknowledgments

   This extension was informed by discussions with Gene Tsudik, and
   various members of the TCPM and TCPCRYPT mailing lists, including
   Christian von Roques.

   This document was prepared using 2-Word-v2.0.template.dot.

Author's Address

   Joe Touch
   USC/ISI
   4676 Admiralty Way
   Marina del Rey, CA 90292
   USA

   Phone: +1 (310) 448-9151
   Email: touch@isi.edu