Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: August 18, 2014

A. Bittau
D. Boneh
M. Hamburg
Stanford University
M. Handley
University College London
D. Mazieres
Q. Slack
Stanford University
February 14, 2014

Cryptographic protection of TCP Streams (tcpcrypt)
draft-bittau-tcp-crypt-04.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

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 chipher. The notation Extract (S, IKM) denotes the output of the extract function with salt S and initial keying material IKM.

- A _collision-resistant pseudo-random function (CPRF)_ is used to generate multiple cryptographic keys from a pseudo-random key, typically the output of the extract function. We use the notation CPRF (K, 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 costs 82 microseconds, while decryption costs 10 milliseconds.

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

3.4. Key exchange protocol

Every machine C has a short-lived public encryption key 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, PK_C
S -> C:  INIT2, sym-cipher, KX_S
```

The parameters are defined as follows:

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

- sym-cipher-list: a list of symmetric cipher suites acceptable to C. These are specified in Table 6.

- N_C: Nonce chosen at random by C.

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

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

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

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

ENC (PK_C, R_S) denotes an encryption of R_S with public key PK_C. R_S and N_S are random values chosen by S. Their lengths are defined in Figure 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    := { pub-cipher-list, sym-cipher-list, sym-cipher }
PRK      := Extract (N_C, { param, PK_C, KX_S, PMS })
```

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 along with 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, K_LEN)
mk[i] := CPRF (mk[i-1], CONST_REKEY, K_LEN)
```

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.

- 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:

- _PRK_ a pseudo-random key.
- _Seq_ the byte position in the stream of Message or Cipher-Text. In tcpcrypt, this is an extended version of TCP's sequence number.
- _Message_ the Message to encrypt. In tcpcrypt, this is TCP's payload.
- _Assoc-Data_ the associated data to be MACed but not encrypted. In tcpcrypt, this contains parts of the TCP header.
- _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.
- _Ciphertext_ the encrypted version of Message.
- _Auth-Tag_ the authenticated encryption authentication tag. In tcpcrypt, this will be the MAC option.

ASM-Decrypt either returns the Valid or Invalid constants, 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.

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

- 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.

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

- Discard the first (Seq-B) bytes on C and begin byte-by-byte XORing the remaining portion with the 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 they might need to retransmit.

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 twelve 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 must be embedded in the TCP handshake. Unfortunately, since the maximum TCP header size is 60 bytes and the basic TCP header fields require 20 bytes, there are at most 40 option payload bytes available, which is not enough to hold the INIT1 and INIT2 messages. Tcpcrypt therefore uses the Data portion of TCP segments (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 state	TCP states for an active opener	TCP states for a passive opener
CLOSED	CLOSED	CLOSED
NEXTK-SENT	SYN-SENT	n/a
HELLO-SENT	SYN-SENT	SYN-RCVD
C-MODE	ESTABLISHED, FIN-WAIT-1	ESTABLISHED, FIN-WAIT-1
LISTEN	n/a	LISTEN
S-MODE	(SYN-RCVD), ESTABLISHED	SYN-RCVD
ENCRYPTING	(SYN-RCVD), ESTABLISHED+	SYN-RCVD, ESTABLISHED+
DISABLED	any	any

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

Table 2

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

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

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

Op designates message types in the abstract protocol, which also correspond to particular suboptions of the TCP CRYPT option, described in Section 4.3, or "MAC" for a valid TCP MAC option, as described in Section 4.4. A segment with SYN=1 and ACK=0 that contains the NEXTK1 suboption will also explicitly or implicitly contain the HELLO suboption; such a segment matches event constraints on either option--e.g., it matches any of the "rcv S/HELLO", "rcv S?/HELLO", and "rcv S/NEXTK1" events. An empty Op matches any segment with the appropriate control bits. A segment MUST contain the TCP MAC option if and only if Op is "MAC".

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

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

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

Figure 1

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

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

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

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

- o Operating systems MUST provide a mechanism for applications to transition to DISABLED from the CLOSED and LISTEN states.
- o A host in the setup phase MUST transition to DISABLED upon receiving any segment without a TCP CRYPT option.
- o A host in the setup phase MUST transition to DISABLED upon receiving any segment with the FIN or RST control bit set.
- o A host in the setup phase MUST transition to DISABLED upon sending a segment with the FIN bit set. (As discussed below, however, a host MUST NOT send a FIN segment from the C-MODE state.)

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

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

A host MUST NOT send a FIN segment from the C-MODE state. The reason is that the remote side can be in the ENCRYPTING state and would thus require the segment to contain a valid MAC, yet a host in C-MODE cannot compute the necessary encryption keys before receiving the INIT2 segment.

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

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

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. However, suboptions in this range are classed as mandatory as far as the protocol is concerned. However, they are not MANDATORY to implement unless otherwise stated, as discussed below.

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

Should a host encounter an unknown opcode greater than or equal to 0xc0 during the setup phase of the protocol, the host MUST transition to the DISABLED state. It SHOULD respond with both a DECLINE suboption and an UNKNOWN suboption specifying the opcode of the unknown mandatory suboption, after which the host MUST NOT send any further CRYPT options.

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

Table 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
0x85	*	IV	E

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.

4.3.1. The HELLO suboption

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

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

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

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

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

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

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

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

4.3.2. The DECLINE suboption

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

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

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

4.3.3. The NEXTK1 and NEXTK2 suboptions

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

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 the following format:

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

Format of the PKCONF suboption

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

```
             0                   1                   2
        bit  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |0|            Algorithm 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-P512.

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 with any given symmetric encryption algorithm and mode. Generally block ciphers will use REKEY while stream ciphers use REKEYSTREAM. We refer to a segment containing either option as a REKEY segment.

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

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

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

Format of the REKEY suboption

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

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

Format of the REKEYSTREAM suboption

A host MAY use REKEY to increment the session key generation number beyond the highest generation it knows the other side to be using. We call this process _initiating_ re-keying. When one host initiates re-keying, the other host MUST increment its key generation number to match, as described 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:

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

- 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
     +-------+-------+-------+-------+
     |         # bytes INIT1         |
     +-------+-------+-------+-------+
     |  INIT1_MAGIC  | # auth enc alg|
     +-------+-------+-------+-------+
     |# bytes of N_C |# byte of PK_C |
     +-------+-------+-------+-------+
     |   authenticated encryption    |
     :          algorithms           :
     +-------+-------+-------+-------+
     |              N_C              |
     :                               :
     +-------+-------+-------+-------+
     |   0   |      type of K_C      |
     +-------+-------+-------+-------+
     |             PK_C              |
     :                               :
     +-------+-------+-------+-------+
```

The INIT1_MAGIC is specified in Table 7. The following values for authenticated sequence mode (ASM) encryption algorithms are defined:

The first entry is mandatory and MUST be supported by all implementations. The sequence number for ASM mode is TCP’s extended 64-bit sequence number offset from the ISN.

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

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

```
Byte     0       1       2       3
     +-------+-------+-------+-------+
     |         # byte INIT2          |
     +-------+-------+-------+-------+
     |  INIT2_MAGIC  |#byte kmaterial|
     +-------+-------+-------+-------+
     |    symmetric cipher suite     |
     +-------+-------+-------+-------+
     |         key material          |
     :                               :
     +-------+-------+-------+-------+
```

Format of the INIT2 suboption

Figure 2

The INIT2_MAGIC is specified in Table 7. The symmetric cipher suite is one selected by the host transmitting the INIT2 segment, which will be playing the "S" role. 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:

```
Byte     0    1       N         M
     +-------+---...---+---...---+
     |# byte |   N_S   |  PK_S   |
     |  N_S  |         |         |
     +-------+---...---+---...---+
```

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.

4.3.10. The IV suboption

The IV suboption is used to hold an initialization vector (IV) when the negotiated encryption mode requires an initialization vector to be transmitted with packets. It MUST NOT be included in transmitted packets except in the ENCRYPTING state when the negotiated encryption mode requires IVs. When the negotiated encryption mode does require IVs, all segments transmitted in ENCRYPTING mode MUST contain an IV suboption.

The IV suboption has the following format:

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

Format of the IV suboption

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

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

4.4. The TCP MAC option

The MAC option is used to authenticate a TCP segment. 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 autheticated 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 0xffffff
   -> 0

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

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

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 hi has wrapped from
   0xffffff -> 0.

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

The two structures, Assoc-Data and Up-Data, are used in ASM mode to calculate the TCP MAC option.

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 P512. 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<1, 4, 16>>
(3) A -> B: A  data<INIT1...>
(4) B -> A: A  data<INIT2...>
(5) A -> B: A  MAC<m> data<...>
```

A wishes to use a key derived from a previous session key, but B does not recognize the session ID or has flushed it from its cache. Therefore, session establishment proceeds as in the first connection, 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_PUBKEY_LOCAL -> When the local host played the "C" role, returns the hosts public key, PK_C. When the local host played the "S" role, returns PK_S if KX_S supports such a value or returns an error otherwise. Hosts MAY return an error after transmitting the first application-level payload bytes (so as to reclaim the memory used to store keys).

TCP_CRYPT_PUBKEY_PEER -> Analogous to TCP_CRYPT_PUBKEY_LOCAL with the roles reversed. (Returns PK_C when the local host played the "S" role, and PK_S, if applicable, when the local host played the "C" role.)

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

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

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

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

TCP_CRYPT_SCONF -> order of preference of symmetric ciphers.

TCP_CRYPT_SUPPORT -> set to 1 if the application is tcpcrypt-aware. set to 2 if the application 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 to 0 for each new connection. A passive opener SHOULD use a default value to 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.

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

7. Acknowledgments

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

8. IANA Considerations

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 Extract: HKDF-Extract-SHA256 CPRF: HKDF-Expand-SHA256 N_C len: 32 bytes R_S len: 48 bytes K_LEN: 32 bytes	0x000100
Cipher: ECDHE-P256 Extract: HKDF-Extract-SHA256 CPRF: HKDF-Expand-SHA256 N_C len: 32 bytes N_S len: 32 bytes K_LEN: 32 bytes	0x000200
Cipher: ECDHE-P512 Extract: HKDF-Extract-SHA256 CPRF: HKDF-Expand-SHA256 N_C len: 32 bytes N_S len: 32 bytes K_LEN: 32 bytes	0x000201

TCP CRYPT algorithm identifiers.

Figure 3

A "tcpcrypt authenticated encryption algorithms" registry needs to be maintained by IANA as per the following table.

Authenticated Encryption	value
AES-128 ASM mode HMAC-SHA2-128 AES-128 ACK MAC	0x00000100
AES-128 ASM mode Poly1305-AES AES-128 ACK MAC	0x00000200
AES-128 ASM mode CMAC-AES-128 AES-128 ACK MAC	0x00000300

TCP CRYPT authenticated encryption algorithms.

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
0x2911	INIT1_MAGIC
0x8310	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

Internet Draft Y. Cheng
draft-cheng-tcpm-fastopen-00.txt J. Chu
Intended status: Experimental S. Radhakrishnan
Creation date: March 7, 2011 A. Jain
Expiration date: September 8, 2011 Google, Inc.

TCP Fast Open

Status of this Memo

Copyright Notice

Abstract

TCP Fast Open (TFO) allows data to be carried in the SYN or SYN-ACK packets and consumed by the receiving end during the initial connection handshake, thus providing a saving of up to one full round trip time (RTT) compared to standard TCP requiring a three-way handshake (3WHS) to complete before data can be exchanged.

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 RFC 2119 [RFC2119]. TFO refers to TCP Fast Open. Client refers to the TCP's active open side and server refers to the TCP's passive open side.

1. Introduction

TCP Fast Open (TFO) enables data to be exchanged safely during TCP connection handshake.

This document describes a design that enables qualified applications to attain a round trip saving while avoiding severe security ramifications. At the core of TFO is a security cookie used by the server side to authenticate a client initiating a TFO connection. The document covers the details of exchanging data during TCP's initial handshake, the protocol for TFO cookies, and potential new security vulnerabilities and their mitigation. It also includes discussions on deployment issues and related proposals. TFO requires extensions to the existing socket API, which will be covered in a separate document.

TFO is motivated by the performance need of today's web applications. Network latency is determined by the round-trip time (RTT) and the number of round trips required to transfer application data. RTT consists of transmission delay and propagation delay. Network bandwidth has grown substantially over the past two decades, much reducing the transmission delay, while propagation delay is largely constrained by the speed of light and has remained unchanged. Therefore reducing the number of round trips has become the most effective way to improve the latency of web applications [CDCM10].

Standard TCP only permits data exchange after 3WHS [RFC793], which introduces one RTT delay to the network latency. For short transfers, e.g., web objects, this additional RTT becomes a significant portion of the network latency [THK98]. One widely deployed solution is HTTP persistent connections. However, this solution is limited since hosts and middle boxes terminate idle TCP connections due to resource constraints. E.g., the Chrome browser keeps TCP connections idle up to 4 minutes but 35% of Chrome HTTP requests are made on new TCP connections.

2. Data In SYN

[RFC793] (section 3.4) already allows data in SYN packets but forbids the receiver to deliver the data to the application until 3WHS is completed. This is because TCP's initial handshake serves to capture

- Old or duplicate SYNs

- SYNs with spoofed IP addresses

TFO allows data to be delivered to the application before 3WHS is completed, thus opening itself to a possible data integrity problem caused by the dubious SYN packets above.

2.1. TCP Semantics and Duplicate SYNs

A past proposal called T/TCP employs a new TCP "TAO" option and connection count to guard against old or duplicate SYNs [RFC1644]. The solution is complex, involving state tracking on per remote peer basis, and is vulnerable to IP spoofing attack. Moreover, it has been shown that even with all the complexity, T/TCP is still not 100% bullet proof. Old or duplicate SYNs may still slip through and get accepted by a T/TCP server [PHRACK98].

Rather than trying to capture all the dubious SYN packets to make TFO 100% compatible with TCP semantics, we've made a design decision early on to accept old SYN packets with data, i.e., to allow TFO for a class of applications that are tolerant of duplicate SYN packets with data, e.g., idempotent or query type transactions. We believe this is the right design trade-off balancing complexity with usefulness. There is a large class of applications that can tolerate dubious transaction requests.

For this reason, TFO MUST be disabled by default, and only enabled explicitly by applications on a per service port basis.

2.2. SYNs with spoofed IP addresses

Standard TCP suffers from the SYN flood attack [RFC4987] because bogus SYN packets, i.e., SYN packets with spoofed source IP addresses can easily fill up a listener's small queue, causing a service port to be blocked completely until timeouts. Secondary damage comes from faked SYN requests taking up memory space. This is normally not an issue today with typical servers having plenty of memory.

TFO goes one step further to allow server side TCP to process and send up data to the application layer before 3WHS is completed. This opens up much more serious new vulnerabilities. Applications serving ports that have TFO enabled may waste lots of CPU and memory resources processing the requests and producing the responses. If the response is much larger than the request, the attacker can mount an amplified reflection attack against victims of choice beyond the TFO server itself.

Numerous mitigation techniques against the regular SYN flood attack exist and have been well documented [RFC4987]. Unfortunately none are applicable to TFO. We propose a server supplied cookie to mitigate most of the security risks introduced by TFO. A more thorough discussion on SYN flood attack against TFO is deferred to the "Security Considerations" section.

3. Protocol Overview

The key component of TFO is the Fast Open Cookie (cookie), a message authentication code (MAC) tag generated by the server. The client requests a cookie in one regular TCP connection, then uses it for future TCP connections to exchange data during 3WHS:

Requesting Fast Open Cookie:

1. The client sends a SYN with a Fast Open Cookie Request option.
2. The server generates a cookie and sends it through the Fast Open Cookie option of a SYN-ACK packet.
3. The client caches the cookie for future TCP Fast Open connections (see below).

Performing TCP Fast Open:

1. The client sends a SYN with Fast Open Cookie option and data.
2. The server validates the cookie:
   a. If the cookie is valid, the server sends a SYN-ACK acknowledging both the SYN and the data. The server then delivers the data to the application.
   b. Otherwise, the server drops the data and sends a SYN-ACK acknowledging only the SYN sequence number.
3. If the server accepts the data in the SYN packet, it may send the response data before the handshake finishes. The max amount is governed by the TCP's congestion control [RFC5681].
4. The client sends an ACK acknowledging the SYN and the server data. If the client's data is not acknowledged, the client retransmits the data in the ACK packet.
5. The rest of the connection proceeds like a normal TCP connection.

The client can perform many TFO operations once it acquires a cookie until the cookie is expired by the server. Thus TFO is useful for applications that have temporal locality on client and server connections.

Requesting Fast Open Cookie in connection 1:

```
   TCP A (Client)                                        TCP B(Server)
   ______________                                        _____________
   CLOSED                                                       LISTEN

#1 SYN-SENT       ----- <SYN,CookieOpt=NIL>  ---------->  SYN-RCVD

#2 ESTABLISHED    <---- <SYN,ACK,CookieOpt=C> ----------  SYN-RCVD
    (caches cookie C)
```

Performing TCP Fast Open in connection 2:

```
   TCP A (Client)                                        TCP B(Server)
   ______________                                        _____________
   CLOSED                                                       LISTEN

#1 SYN-SENT       ----- <SYN=x,CookieOpt=C,DATA_A> ---->  SYN-RCVD

#2 ESTABLISHED    <---- <SYN=y,ACK=x+len(DATA_A)+1> ----  SYN-RCVD

#3 ESTABLISHED    <---- <ACK=x+len(DATA_A)+1,DATA_B>----  SYN-RCVD

#4 ESTABLISHED    ----- <ACK=y+1>--------------------> ESTABLISHED

#5 ESTABLISHED    --- <ACK=y+len(DATA_B)+1>----------> ESTABLISHED
```

4. Protocol Details

4.1. Fast Open Cookie

The Fast Open Cookie is invented to mitigate new security vulnerabilities in order to enable data exchange during handshake. The cookie is a message authentication code tag generated by the server and is opaque to the client; the client simply caches the cookie and passes it back on subsequent SYN packets to open new connections. The server can expire the cookie at any time to enhance security.

4.1.1. TCP Options

Fast Open Cookie Option

The server uses this option to grant a cookie to the client in the SYN-ACK packet; the client uses it to pass the cookie back to the server in the SYN packet.

```
                                +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                                |      Kind     |    Length     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                            Cookie                             ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Kind            1 byte: constant TBD (assigned by IANA)
Length          1 byte: range 6 to 18 (bytes); limited by
                        remaining space in the options field.
                        The number MUST be even.
Cookie          4 to 16 bytes (Length - 2)
```

Options with invalid Length values or without SYN flag set MUST be ignored. The minimum Cookie size is 4 bytes. Although the diagram shows a cookie aligned on 32-bit boundaries, that is not required.

Fast Open Cookie Request Option

The client uses this option in the SYN packet to request a cookie from a TFO-enabled server

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Kind     |    Length     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
Kind            1 byte: same as the Fast Open Cookie option
Length          1 byte: constant 2. This distinguishes the option from
                        the Fast Open cookie option.
```

Options with invalid Length values, without SYN flag set, or with ACK flag set MUST be ignored.

4.1.2. Server Cookie Handling

The server is in charge of cookie generation and authentication. The cookie SHOULD be a message authentication code tag with the following properties:

1. The cookie authenticates the client's (source) IP address of the SYN packet. The IP address can be an IPv4 or IPv6 address.

2. The cookie can only be generated by the server and can not be fabricated by any other parties including the client.

3. The cookie expires after a certain amount of time. The reason is detailed in the "Security Consideration" section. This can be done by either periodically changing the server key used to generate cookies or including a timestamp in the cookie.

4. The generation and verification are fast relative to the rest of SYN and SYN-ACK processing.

5. A server may encode other information in the cookie, and allow more than one valid cookie per client at any given time. But this is all server implementation dependent and transparent to the client. A client only needs to remember one valid cookie per server IP.

The server supports the cookie generation and verification operations:

- GetCookie(IP_Address): returns a (new) cookie

- IsCookieValid(IP_Address, Cookie): checks if the cookie is valid, i.e., it has not expired and it authenticates the client IP address.

Example Implementation: a simple implementation is to use AES_128 to encrypt the IPv4 (with padding) or IPv6 address and truncate to 64 bits. The server can periodically update the key to expire the cookies. AES encryption on recent processors is fast and takes only a few hundred nanoseconds.

4.1.3. Client Cookie Handling

The client MUST cache cookies from different servers for later Fast Open connections. For a multi-homed client, the cookies are both client and server IP dependent. Beside the cookie, we RECOMMEND that the client caches the MSS and RTT to the server to enhance performance.

The MSS advertised by the server is stored in the cache to determine the maximum amount of data that can be supported in the SYN packet. This information is needed because data is sent before the server announces its MSS in the SYN-ACK packet. Without this information, the data size in the SYN packet is limited to the default MSS of 536 bytes [RFC1122].

Caching RTT allows seeding a more accurate SYN timeout than the default value [RFC2988]. This lowers the performance penalty if the network or the server drops the SYN packets with data or the cookie options (See "Reliability and Deployment Issues" section below).

The cache replacement algorithm is not specified and is left for the implementations.

4.2. Fast Open Protocol

One predominant requirement of TFO is to be fully compatible with existing TCP implementations, both on the client and the server sides.

The server keeps two variables per listening port:

FastOpenEnabled: default is off. It MUST be turned on explicitly by the application. When this flag is off, the server does not perform any TFO related operations and MUST ignore all cookie options.

PendingFastOpenRequests: tracks number of TFO connections in SYN-RCVD state. If this variable goes over the system limit, the server SHOULD set FastOpenEnabled off. This variable is used for defending some vulnerabilities discussed in the "Security Considerations" section.

The server keeps a FastOpened flag per TCB to mark if a connection has successfully performed a TFO.

4.2.1. Fast Open Cookie Request

Any client attempting TFO MUST first request a cookie from the server with the following steps:

1. The client sends a SYN packet with a Fast Open Cookie Request

option.

2. The server SHOULD respond with a SYN-ACK based on the procedures in the "Server Cookie Handling" section. This SYN-ACK SHOULD contain a Fast Open Cookie option if the server currently supports TFO for this listener port.

3. If the SYN-ACK contains a valid Fast Open Cookie option, the client replaces the cookie and other information as described in the "Client Cookie Handling" section. Otherwise, if the SYN-ACK is first seen, i.e.,not a (spurious) retransmission, the client MAY remove the server information from the cookie cache. If the SYN-ACK is a spurious retransmission without valid Fast Open Cookie Option, the client does nothing to the cookie cache for the reasons below.

The network or servers may drop the SYN or SYN-ACK packets with the new cookie options which causes SYN or SYN-ACK timeouts. We RECOMMEND both the client and the server retransmit SYN and SYN-ACK without the cookie options on timeouts. This ensures the connections of cookie requests will go through and lowers the latency penalties (of dropped SYN/SYN-ACK packets). The obvious downside for maximum compatibility is that any regular SYN drop will fail the cookie. We also RECOMMEND the client to record servers that failed to respond to cookie requests and only attempt another cookie request after certain period.

4.2.2. TCP Fast Open

Once the client obtains the cookie from the target server, the client can perform subsequent TFO connections until the cookie is expired by the server. The nature of TCP sequencing makes the TFO specific changes relatively small in addition to [RFC793].

Client: Sending SYN

To open a TFO connection, the client MUST have obtained the cookie from the server:

1. Send a SYN packet.

   a. If the SYN packet does not have enough option space for the Fast Open Cookie option, abort TFO and fall back to regular 3WHS.

   b. Otherwise, include the Fast Open Cookie option with the cookie of the server.Include any data up to the cached server MSS or default 536 bytes.

2. Advance to SYN-SENT state and update SND.NXT to include the data accordingly.

3. If RTT is available from the cache, seed SYN timer according to [RFC2988].

To deal with network or servers dropping SYN packets with payload or unknown options, when the SYN timer fires, the client SHOULD retransmit a SYN packet without data and Fast Open Cookie options.

Server: Receiving SYN and responding with SYN-ACK

Upon receiving the SYN packet with Fast Open Cookie option:

1. If the cookie is invalid, i.e., the cookie does not authenticate the source IP address of the SYN packet, send a SYN-ACK packet acknowledging only the SYN sequence. In addition, include a Fast Open Cookie Option with a new cookie. Go to step 7.

2. If PendingFastOpenRequests is over the system limit, reset FastOpenEnabled flag and send a SYN-ACK acknowledging only the SYN sequence. Go to step 7.

3. Send the SYN-ACK packet acknowledging the SYN and data sequence. The server MAY include data in the SYN-ACK packet.

4. Buffer the data and notify the application.

5. Set FastOpened flag and increment PendingFastOpenRequests.

6. The server MAY send more data packets before the handshake completes. The maximum amount is ruled by the initial congestion window and the receiver window [RFC3390].

7. Advance to the SYN-RCVD state.

If the SYN-ACK timer fires, the server SHOULD retransmit a SYN-ACK packet without data and Fast Open Cookie options for compatibility reasons.

Client: Receiving SYN-ACK

The client SHOULD perform the following steps upon receiving the SYN-ACK:

1. Update the cookie cache if the SYN-ACK has a Fast Open Cookie Option.

2. Send an ACK packet. Set acknowledgment number to RCV.NXT and

include the data after SND.UNA if data is available

3. Advance to the ESTABLISHED state

Note there is no latency penalty if the server does not acknowledge the data in the original SYN packet. The client will retransmit it in the ACK packet. The data exchange will start after the handshake like a regular TCP connection.

Server: Receiving ACK

Upon receiving an ACK acknowledging the SYN sequence, the server decrements PendingFastOpenRequests and advances to the ESTABLISHED state. No special handling is required further.

5. Reliability and Deployment Issues

Network or Hosts Dropping SYN packets with data or unknown options

A study [MAF04] found that some middle-boxes and end-hosts may drop packets with unknown TCP options incorrectly. Another study [LANGLEY06] found that 6% of the probed paths on the Internet drop SYN packets with data. The TFO protocol deals with this problem by retransmitting SYN without data or cookie options and we recommend tracking these servers in the client.

Server Farms

A common server-farm setup is to have many physical hosts behind a load-balancer sharing the same server IP. The load-balancer forwards new TCP connections to different physical hosts based on certain load-balancing algorithms. For TFO to work, the physical hosts need to share the same key and update the key at about the same time.

Network Address Translation (NAT)

The hosts behind NAT sharing same IP address will get the same cookie to the same server. This will not prevent TFO from working. But on some carrier-grade NAT configurations where every new TCP connection from the same physical host uses a different public IP address, TFO does not provide latency benefit. However, there is no performance penalty either as described in Section "Client receiving SYN-ACK".

6. Security Considerations

The Fast Open cookie stops an attacker from trivially flooding spoofed SYN packets with data to burn server resources or to mount an amplified reflection attack on random hosts. The server can defend against spoofed SYN floods with invalid cookies using existing techniques [RFC4987].

However, the attacker may still obtain cookies from some compromised hosts, then flood spoofed SYN with data and "valid" cookies (from these hosts or other vantage points). With DHCP, it's possible to obtain cookies of past IP addresses without compromising any host. Below we identify two new vulnerabilities of TFO and describe the countermeasures.

6.1. Server Resource Exhaustion Attack by SYN Flood with Valid Cookies

Like regular TCP handshakes, TFO is vulnerable to such an attack. But the potential damage can be much more severe. Besides causing temporary disruption to service ports under attack, it may exhaust server CPU and memory resources.

For this reason it is crucial for the TFO server to limit the maximum number of total pending TFO connection requests, i.e., PendingFastOpenRequests. When the limit is exceeded, the server temporarily disables TFO entirely as described in "Server Cookie Handling". Then subsequent TFO requests will be downgraded to regular connection requests, i.e., with the data dropped and only SYN acknowledged. This allows regular SYN flood defense techniques [RFC4987] like SYN-cookies to kick in and prevent further service disruption.

There are other subtle but important differences in the vulnerability between TFO and regular TCP handshake. Before the SYN flood attack broke out in the late '90s, typical listener's max qlen was small, enough to sustain the highest expected new connection rate and the average RTT for the SYN-ACK packets to be acknowledged in time. E.g., if a server is designed to handle at most 100 connection requests per second, and the average RTT is 100ms, a max qlen on the order of 10 will be sufficient.

This small max qlen made it very easy for any attacker, even equipped with just a dailup modem to the Internet, to cause major disruptions to a web site by simply throwing a handful of "SYN bombs" at its victim of choice. But for this attack scheme to work, the attacker must pick a non-responsive source IP address to spoof with. Otherwise the SYN-ACK packet will trigger TCP RST from the host whose IP address has been spoofed, causing corresponding connection to be removed from the server's listener queue hence defeating the attack. In other words, the main damage of SYN bombs against the standard TCP stack is not directly from the bombs themselves costing TCP processing overhead or host memory, but rather from the spoofed SYN packets filling up the often small listener's queue.

On the other hand, TFO SYN bombs can cause damage directly if admitted without limit into the stack. The RST packets from the spoofed host will fuel rather than defeat the SYN bombs as compared to the non-TFO case, because the attacker can flood more SYNs with data to cost more data processing resources. For this reason, a TFO server needs to monitor the connections in SYN-RCVD being reset in addition to imposing a reasonable max qlen. Implementations may combine the two, e.g., by continuing to account for those connection requests that have just been reset against the listener's PendingFastOpenRequests until a timeout period has passed.

Limiting the maximum number of pending TFO connection requests does make it easy for an attacker to overflow the queue, causing TFO to be disabled. We argue that causing TFO to be disabled is unlikely to be of interest to attackers because the service will remain intact without TFO hence there is hardly any real damage.

6.2. Amplified Reflection Attack to Random Host

Limiting PendingFastOpenRequests with a system limit can be done without Fast Open Cookies and would protect the server from resource exhaustion. It would also limit how much damage an attacker can cause through an amplified reflection attack from that server. However, it would still be vulnerable to an amplified reflection attack from a large number of servers. An attacker can easily cause damage by tricking many servers to respond with data packets at once to any spoofed victim IP address of choice.

With the use of Fast Open Cookies, the attacker would first have to steal a valid cookie from its target victim. This likely requires the attacker to compromise the victim host or network first.

The attacker here has little interest in mounting an attack on the victim host that has already been compromised. But she may be motivated to disrupt the victim's network. Since a stolen cookie is only valid for a single server, she has to steal valid cookies from a large number of servers and use them before they expire to cause sufficient damage without triggering the defense in the previous section.

One can argue that if the attacker has compromised the target network or hosts, she could perform a similar but simpler attack by injecting bits directly. The degree of damage will be identical, but TFO-specific attack allows the attacker to remain anonymous and disguises the attack as from other servers.

The best defense is for the server to not respond with data until handshake finishes, i.e., disallow step 6 in "Server receiving SYN-

ACK" section. In this case the risk of amplification reflection attack is completely eliminated, but the potential latency saving from TFO may diminish if the server application produces responses earlier before the handshake completes.

7. Related Work

7.1. T/TCP

TCP Extensions for Transactions [RFC1644] attempted to bypass the three-way handshake, among other things, hence shared the same goal but also the same set of issues as TFO. It focused most of its effort battling old or duplicate SYNs, but paid no attention to security vulnerabilities it introduced when bypassing 3WHS. Its TAO option and connection count, besides adding complexity, require the server to keep state per remote host, while still leaving it wide open for attacks. It is trivial for an attacker to fake a CC value that will pass the TAO test. Unfortunately, in the end its scheme is still not 100% bullet proof as pointed out by [PHRACK98].

As stated earlier, we take a practical approach to focus TFO on the security aspect, while allowing old, duplicate SYN packets with data after recognizing that 100% TCP semantics is likely infeasible. We believe this approach strikes the right tradeoff, and makes TFO much simpler and more appealing to TCP implementers and users.

7.2. Common Defenses Against SYN Flood Attacks

TFO is still vulnerable to SYN flood attacks just like normal TCP handshakes, but the damage may be much worse, thus deserves a careful thought.

There have been plenty of studies on how to mitigate attacks from regular SYN flood, i.e., SYN without data [RFC4987]. But from the stateless SYN-cookies to the stateful SYN Cache, none can preserve data sent with SYN safely while still providing an effective defense.

The best defense may be to simply disable TFO when a host is suspected to be under a SYN flood attack, e.g., the SYN backlog is filled. Once TFO is disabled, normal SYN flood defenses can be applied. The "Security Consideration" section contains a thorough discussion on this topic.

7.3. TCP Cookie Transaction (TCPCT)

TCPCT [RFC6013] eliminates server state during initial handshake and defends spoofing DoS attacks. Like TFO, TCPCT allows SYN and SYN-ACK packets to carry data. However, TCPCT and TFO are designed for different goals and they are not compatible.

The TCPCT server does not keep any connection state during the handshake, therefore the server application needs to consume the data in SYN and (immediately) produce the data in SYN-ACK before sending SYN-ACK. Otherwise the application’s response has to wait until handshake completes. In contrary, TFO allows server to respond data during handshake. Therefore for many request-response style applications, TCPCT may not achieve same latency benefit as TFO.

Without state kept on the server side, TCPCP relies on the client side to retransmit the SYN request with data in order to recover from possible loss of packet from server response. This may cause a lot more dubious connection requests. It also limits the response to only one packet, to fit completely within the SYN-ACK packet. For some TCP applications, in particular web applications, this does not provide enough latency benefit by sending one data packet one RTT earlier.

8. IANA Considerations

The Fast Open Cookie Option and Fast Open Cookie Request Option define no new namespace. The options require IANA allocate one value from the TCP option Kind namespace.

9. Acknowledgements

The authors would like to thank Tom Herbert, Adam Langley, Roberto Peon, Mathew Mathis, and Barath Raghavan for their insightful comments.

10. References

10.1. Normative References

[RFC793] Postel, J. "Transmission Control Protocol", RFC 793, September 1981.

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

[RFC2988] Paxson, V. and M. Allman, "Computing TCP's Retransmission Timer", RFC 2988, November 2000.

[RFC5681] Allman, M., Paxson, V. and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.

10.2. Informative References

[RFC1644] Braden, R., "T/TCP -- TCP Extensions for Transactions Functional Specification", RFC 1644, July 1994.

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

[RFC6013] Simpson, W., "TCP Cookie Transactions (TCPCT)", RFC6013, January 2011.

[CDCM10] Chu, J., Dukkipati, N., Cheng, Y. and M. Mathis, "Increasing TCP's Initial Window", Internet-Draft draft-ietf-tcpm-initcwnd-00.txt (work in progress), October 2010.

[THK98] Touch, J., Heidemann, J., Obraczka, K., "Analysis of HTTP Performance", USC/ISI Research Report 98-463. December 1998.

[PHRACK98] "T/TCP vulnerabilities", Phrack Magazine, Volume 8, Issue 53 artical 6. July 8, 1998. URL http://www.phrack.com/issues.html?issue=53&id=6

[MAF04] Medina, A., Allman, M., and S. Floyd, "Measuring Interactions Between Transport Protocols and Middleboxes", Proceedings 4th ACM SIGCOMM/USENIX Conference on Internet Measurement, October 2004.

[LANGLEY06] Langley, A, "Probing the viability of TCP extensions", URL http://www.imperialviolet.org/binary/ecntest.pdf

Author's Addresses

Yuchung Cheng
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043, USA
EMail: ycheng@google.com

H.K. Jerry Chu
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043, USA
EMail: hkchu@google.com

Sivasankar Radhakrishnan
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043, USA
EMail: sivasankar@google.com

Arvind Jain
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043, USA
EMail: arvind@google.com

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.

TCP Maintenance and Minor Extensions (tcpm)
Internet-Draft
Obsoletes: 1948 (if approved)
Updates: 793 (if approved)
Intended status: Standards Track
Expires: July 7, 2011

F. Gont
UTN/FRH
S. Bellovin
Columbia University

January 3, 2011

Defending Against Sequence Number Attacks
draft-gont-tcpm-rfc1948bis-00.txt

Abstract

This document specifies an algorithm for the generation of TCP Initial Sequence Numbers (ISNs), such that the chances of an off-path attacker of guessing the sequence numbers in use by a target connection are reduced. This document is a revision of RFC 1948, and takes the ISN generation algorithm originally proposed in that document to Standards Track.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

During the last 25 years, the Internet has experienced a number of off-path attacks against TCP connections. These attacks have ranged from trust relationships exploitation to Denial of Service attacks [CPNI-TCP]. Discusion of some of these attacks dates back to at least 1985, when Morris [Morris1985] described a form of attack based on guessing what sequence numbers TCP [RFC0793] will use for new connections.

In 1996, RFC 1948 [RFC1948] proposed an algorithm for the selection of TCP Initial Sequence Numbers (ISNs), such that the chances of an off-path attacker of guessing valid sequence numbers are reduced. With the aforementioned algorithm, such attacks would remain possible if and only if the Bad Guy already had the ability to launch even more devastating attacks.

This document is a revision of RFC 1948, and takes the ISN generation algorithm originally proposed in that document to Standards Track.

Section 2 provides a brief discussion of the requirements for a good ISN generation algorithm. Section 3 specifies a good ISN randomization algorithm. Finally, Appendix A provides a discussion of the trust-relationship exploitation attacks that originally motivated the publication of RFC 1948 [RFC1948].

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. Generation of Initial Sequence Numbers

RFC 793 [RFC0793] suggests that the choice of the Initial Sequence Number of a connection is not arbitrary, but aims to reduce the chances of a stale segment from being accepted by a new incarnation of a previous connection. RFC 793 [RFC0793] suggests the use of a global 32-bit ISN generator that is incremented by 1 roughly every 4 microseconds.

It is interesting to note that, as a matter of fact, protection against stale segments from a previous incarnation of the connection is enforced by preventing the creation of a new incarnation of a previous connection before 2*MSL have passed since a segment corresponding to the old incarnation was last seen. This is accomplished by the TIME-WAIT state, and TCP’s "quiet time" concept (see Appendix B of [RFC1323]).

Based on the assumption that ISNs are monotonically-increasing across connections, many stacks (e.g., 4.2BSD-derived) use the ISN of an incomming SYN segment to perform "heuristics" that enable the creation of a new incarnation of a connection while the previous incarnation is still in the TIME-WAIT state (see pp. 945 of [Wright1994]). This avoids an interoperability problem that may arise when a systems establishes connections to a specific TCP end-point at a high rate [Silbersack2005].

Unfortunately, the ISN generator described in [RFC0793] makes it trivial for an off-path attacker to predict the ISN that a TCP will use for new connections, thus allowing a variety of attacks against TCP connections [CPNI-TCP]. One of the possible attacks that took advantage of weak sequence numbers was first described in [Morris1985], and its exploitation was widely publicized about 10 years later [Shimomura1995]. [CERT2001] and [USCERT2001] are advisories about the security implications of weak ISN generators. [Zalewski2001] and [Zalewski2002] contain a detailed analysis of ISN generators, and a survey of the algorithms in use by popular TCP implementations.

Simple randomization of the TCP Initial Sequence Numbers would mitigate those attacks that require an attacker to guess valid sequence numbers. However, it would also break the 4.4BSD "heuristics" to accept a new incoming connection when there is a previous incarnation of that connection in the TIME-WAIT state [Silbersack2005].

We can prevent sequence number guessing attacks by giving each connection -- that is, each 4-tuple of (localip, localport, remoteip, remoteport) -- a separate sequence number space. Within each space, the initial sequence number is incremented according to [RFC0793]; however, there is no obvious relationship between the numbering in different spaces.

The obvious way to do this is to maintain state for dead connections, and the easiest way to do that is to change the TCP state transition diagram so that both ends of all connections go to TIME-WAIT state. That would work, but it’s inelegant and consumes storage space. Instead, we propose an improvement to the TCP ISN generation algorithm.

3. Proposed Initial Sequence Number (ISN) generation algorithm

TCP SHOULD generate its Initial Sequence Numbers with the expression:

ISN = M + F(localip, localport, remoteip, remoteport)

where M is the 4 microsecond timer, and F is a pseudorandom function (PRF) of the connection-id. It is vital that F not be computable from the outside, or an attacker could still guess at sequence numbers from the initial sequence number used for some other connection. The PRF could be implemented as a cryptographic hash of the concatenation of the connection-id and some secret data; SHA-256 [FIPS-SHS] would be a good choice for the hash function. The secret data can either be a true random number [RFC4086], or it can be the combination of some per-host secret and the boot time of the machine. The boot time is included to ensure that the secret is changed on occasion.

Note that the secret cannot easily be changed on a live machine. Doing so would change the initial sequence numbers used for reincarnated connections; to maintain safety, either dead connection state must be kept or a quiet time observed for two maximum segment lifetimes after such a change.

4. Security Considerations

Good sequence numbers are not a replacement for cryptographic authentication, such as that provided by IPsec [RFC4301]. At best, they're a palliative measure.

If random numbers are used as the sole source of the secret, they MUST be chosen in accordance with the recommendations given in [RFC4086].

A security consideration that should be made about the algorithm proposed in this document is that it might allow an attacker to count the number of systems behind a Network Address Translator (NAT) [RFC3022]. Depending on the ISN generators implemented by each of the systems behind the NAT, an attacker might be able to count the number of systems behind a NAT by establishing a number of TCP connections (using the public address of the NAT) and indentifying the number of different sequence number "spaces". [I-D.gont-behave-nat-security] discusses how this and other information leakages at NATs could be mitigated.

An eavesdropper who can observe the initial messages for a connection can determine its sequence number state, and may still be able to launch sequence number guessing attacks by impersonating that connection. However, such an eavesdropper can also hijack existing connections [Joncheray1995], so the incremental threat isn't that high. Still, since the offset between a fake connection and a given real connection will be more or less constant for the lifetime of the secret, it is important to ensure that attackers can never capture such packets. Typical attacks that could disclose them include both eavesdropping and the variety of routing attacks discussed in [Bellovin1989].

[CPNI-TCP] contains a discussion of all the currently-known attacks that require an attacker to know or be able to guess the TCP sequence numbers in use by the target connection.

5. IANA Considerations

This document has no actions for IANA.

6. Acknowledgements

Matt Blaze and Jim Ellis contributed some crucial ideas to RFC 1948, on which this document is based. Frank Kastenholz contributed constructive comments to that memo.

The authors of this document woul like to thank (in chronological order) Alfred Hoenes for providing valuable comments on earlier versions of this document.

Fernando Gont would like to thank the United Kingdom’s Centre for the Protection of National Infrastructure (UK CPNI) for their continued support.

7. References

7.1. Normative References

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

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

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

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

[RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness

Requirements for Security", BCP 106, RFC 4086, June 2005.

7.2. Informative References

[Bellovin1989]
Morris, R., "Security Problems in the TCP/IP Protocol Suite", Computer Communications Review, vol. 19, no. 2, pp. 32-48, 1989.

[CERT2001]
CERT, "CERT Advisory CA-2001-09: Statistical Weaknesses in TCP/IP Initial Sequence Numbers", http://www.cert.org/advisories/CA-2001-09.html, 2001.

[CPNI-TCP]
CPNI, "Security Assessment of the Transmission Control Protocol (TCP)", http://www.cpni.gov.uk/Docs/tn-03-09-security-assessment-TCP.pdf, 2009.

[FIPS-SHS]
FIPS, "Secure Hash Standard (SHS)", Federal Information Processing Standards Publication 180-3, 2008, available at: http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf.

[I-D.gont-behave-nat-security]
Gont, F. and P. Srisuresh, "Security implications of Network Address Translators (NATs)", draft-gont-behave-nat-security-03 (work in progress), October 2009.

[Joncheray1995]
Joncheray, L., "A Simple Active Attack Against TCP", Proc. Fifth Usenix UNIX Security Symposium, 1995.

[Morris1985]
Morris, R., "A Weakness in the 4.2BSD UNIX TCP/IP Software", CSTR 117, AT&T Bell Laboratories, Murray Hill, NJ, 1985.

[RFC0854] Postel, J. and J. Reynolds, "Telnet Protocol Specification", STD 8, RFC 854, May 1983.

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, November 1987.

[RFC1948] Bellovin, S., "Defending Against Sequence Number Attacks", RFC 1948, May 1996.

[RFC3022] Srisuresh, P. and K. Egevang, "Traditional IP Network Address Translator (Traditional NAT)", RFC 3022, January 2001.

[RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, July 2005.

[RFC4251] Ylonen, T. and C. Lonvick, "The Secure Shell (SSH) Protocol Architecture", RFC 4251, January 2006.

[RFC4301] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, December 2005.

[RFC4954] Siemborski, R. and A. Melnikov, "SMTP Service Extension for Authentication", RFC 4954, July 2007.

[RFC5321] Klensin, J., "Simple Mail Transfer Protocol", RFC 5321, October 2008.

[RFC5936] Lewis, E. and A. Hoenes, "DNS Zone Transfer Protocol (AXFR)", RFC 5936, June 2010.

[Shimomura1995]
Shimomura, T., "Technical details of the attack described by Markoff in NYT", http://www.gont.com.ar/docs/post-shimomura-usenet.txt, Message posted in USENET's comp.security.misc newsgroup, Message-ID: <3g5gkl$5j1@ariel.sdsc.edu&gt, 1995.

[Silbersack2005]
Silbersack, M., "Improving TCP/IP security through randomization without sacrificing interoperability.", EuroBSDCon 2005 Conference .

[USCERT2001]
US-CERT, "US-CERT Vulnerability Note VU#498440: Multiple TCP/IP implementations may use statistically predictable initial sequence numbers", http://www.kb.cert.org/vuls/id/498440, 2001.

[Wright1994]
Wright, G. and W. Stevens, "TCP/IP Illustrated, Volume 2: The Implementation", Addison-Wesley, 1994.

[Zalewski2001]
Zalewski, M., "Strange Attractors and TCP/IP Sequence Number Analysis",

http://lcamtuf.coredump.cx/oldtcp/tcpseq.html, 2001.

[Zalewski2002]
Zalewski, M., "Strange Attractors and TCP/IP Sequence Number Analysis - One Year Later", http://lcamtuf.coredump.cx/newtcp/, 2002.

Appendix A. Address-based trust relationship exploitation attacks

This section discusses the trust-relationship exploitation attack that originally motivated the publication of RFC 1948 [RFC1948]. It should be noted that while RFC 1948 focused its discussion of address-based trust relationship exploitation attacks on Telnet [RFC0854] and the various UNIX "r" commands, both Telnet and the various "r" commands have since been largely replaced by secure counter-parts (such as SSH [RFC4251]) for the purpose of remote login and remote command execution. Nevertheless, address-based trust relationships are still employed nowadays in some scenarios. For example, some SMTP [RFC5321] deployments still authenticate their users by means of their IP addresses, even when more appropriate authentication mechanisms are available [RFC4954]. Another example is the authentication of DNS secondary servers [RFC1034] by means of their IP addresses for allowing DNS zone transfers [RFC5936], or any other access control mechanism based on IP addresses.

In 1985, Morris [Morris1985] described a form of attack based on guessing what sequence numbers TCP [RFC0793] will use for new connections. Briefly, the attacker gags a host trusted by the target, impersonates the IP address of the trusted host when talking to the target, and completes the 3-way handshake based on its guess at the next initial sequence number to be used. An ordinary connection to the target is used to gather sequence number state information. This entire sequence, coupled with address-based authentication, allows the attacker to execute commands on the target host.

Clearly, the proper solution for these attacks is cryptographic authentication [RFC4301] [RFC4120] [RFC4251].

The following subsections provide technical details for the trust relationship exploitation attack described by Morris [Morris1985].

A.1. Blind TCP connection-spoofing

In order to understand the particular case of sequence number guessing, one must look at the 3-way handshake used in the TCP open sequence [RFC0793]. Suppose client machine A wants to talk to rsh server B. It sends the following message:

A->B: SYN, ISNa

That is, it sends a packet with the SYN ("synchronize sequence number") bit set and an initial sequence number ISNa.

B replies with

B->A: SYN, ISNb, ACK(ISNa)

In addition to sending its own initial sequence number, it acknowledges A’s. Note that the actual numeric value ISNa must appear in the message.

A concludes the handshake by sending

A->B: ACK(ISNb)

RFC 793 [RFC0793] specifies that the 32-bit counter be incremented by 1 in the low-order position about every 4 microseconds. Instead, Berkeley-derived kernels traditionally incremented it by a constant every second, and by another constant for each new connection. Thus, if you opened a connection to a machine, you knew to a very high degree of confidence what sequence number it would use for its next connection. And therein lied the vulnerability.

The attacker X first opens a real connection to its target B -- say, to the mail port or the TCP echo port. This gives ISNb. It then impersonates A and sends

Ax->B: SYN, ISNx

where "Ax" denotes a packet sent by X pretending to be A.

B’s response to X’s original SYN (so to speak)

B->A: SYN, ISNb’, ACK(ISNx)

goes to the legitimate A, about which more anon. X never sees that message but can still send

Ax->B: ACK(ISNb’)

using the predicted value for ISNb’. If the guess is right -- and usually it will be, if the sequence numbers are weak -- B’s rsh server thinks it has a legitimate connection with A, when in fact X is sending the packets. X can’t see the output from this session, but it can execute commands as more or less any user -- and in that case, the game is over and X has won.

There is a minor difficulty here. If A sees B’s message, it will realize that B is acknowledging something it never sent, and will send a RST packet in response to tear down the connection. There are a variety of ways to prevent this; the easiest is to wait until the real A is down (possibly as a result of enemy action, of course). In actual practice, X can gag A by exploiting a very common implementation bug; this is described in the next subsection.

A.2. An old BSD bug

As mentioned in the previous sub-section, attackers performing a trust relationship exloitation attack may want to "gag" the trusted machine first. While a number of strategies are possible, most of the attacks detected in the wild relied on an implementation bug.

When SYN packets are received for a connection, the receiving system creates a new TCB in SYN-RCVD state. To avoid overconsumption of resources, 4.2BSD-derived systems permit only a limited number of TCBs in this state per connection. Once this limit is reached, future SYN packets for new connections are discarded; it is assumed that the client will retransmit them as needed.

When a packet is received, the first thing that must be done is a search for the TCB for that connection. If no TCB is found, the kernel searches for a "wild card" TCB used by servers to accept connections from all clients. Unfortunately, in many kernels this code was invoked for any incoming packets, not just for initial SYN packets. If the SYN-RCVD queue was full for the wildcard TCB, any new packets specifying just that host and port number were discarded, even if they weren’t SYN packets.

To gag a host, then, the attacker sent a few dozen SYN packets to the rlogin port from different port numbers on some non-existent machine. This filled up the SYN-RCVD queue, while the SYN+ACK packets went off to the bit bucket. The attack on the target machine then appeared to come from the rlogin port on the trusted machine. The replies -- the SYN+ACKs from the target -- were perceived as packets belonging to a full queue, and were dropped silently. This could have been avoided if the full queue code checked for the ACK bit, which could not legally be on for legitimate open requests (if it was on, an RST should be sent in response).

Appendix B. Changes from previous versions of the document

B.1. Changes from RFC 1948

- New document aims at Standards Track (rather than Informaitonal).

- The discussion of address-based trust relationship attacks was updated and moved to an Appendix.

- The recommended hash algorithm has been changed to SHA-256 [FIPS-SHS], in response to the security concerns for MD5 [RFC1321].

- Formal requirements ([RFC2119]) are specified.

Authors' Addresses

Fernando Gont
Universidad Tecnologica Nacional / Facultad Regional Haedo
Evaristo Carriego 2644
Haedo, Provincia de Buenos Aires 1706
Argentina

Phone: +54 11 4650 8472
Email: fernando@gont.com.ar
URI: http://www.gont.com.ar

Steven M. Bellovin
Columbia University
1214 Amsterdam Avenue
MC 0401
New York, NY 10027
US

Phone: +1 212 939 7149
Email: bellovin@acm.org

Network Working Group
Internet-Draft
Obsoletes: 3782 (if approved)
Intended status: Standards Track
Expires: September 15, 2011

T. Henderson
Boeing
S. Floyd
ICSI
A. Gurtov
HIIT
Y. Nishida
WIDE Project

March 14, 2011

The NewReno Modification to TCP's Fast Recovery Algorithm

draft-ietf-tcpm-rfc3782-bis-01.txt

Abstract

RFC 5681 [RFC5681] documents the following four intertwined TCP congestion control algorithms: Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery. RFC 5681 explicitly allows certain modifications of these algorithms, including modifications that use the TCP Selective Acknowledgement (SACK) option [RFC2883], and modifications that respond to "partial acknowledgments" (ACKs which cover new data, but not all the data outstanding when loss was detected) in the absence of SACK. This document describes a specific algorithm for responding to partial acknowledgments, referred to as NewReno. This response to partial acknowledgments was first proposed by Janey Hoe in [Hoe95].

The purpose of this revision from [RFC3782] is to make errata changes and to adopt a proposal from Yoshifumi Nishida to slightly increase the minimum window size after Fast Recovery from one to two segments, to improve performance when the receiver uses delayed acknowledgments.

Status of this Memo

This Internet-Draft will expire on September 15, 2011.

Copyright Notice

1. Introduction

For the typical implementation of the TCP Fast Recovery algorithm described in [RFC5681] (first implemented in the 1990 BSD Reno release, and referred to as the Reno algorithm in [FF96]), the TCP data sender only retransmits a packet after a retransmit timeout has occurred, or after three duplicate acknowledgments have arrived triggering the Fast Retransmit algorithm. A single retransmit timeout might result in the retransmission of several data packets, but each invocation of the Fast Retransmit algorithm in RFC 5681 leads to the retransmission of only a single data packet.

Two problems arise with Reno TCP when multiple packet losses occur in a single window. First, Reno will often take a timeout, as has been documented in [Hoe95]. Second, even if a retransmission timeout is avoided, multiple fast retransmits and window reductions can occur, as documented in [F94]. When multiple packet losses occur, if the SACK option [RFC2883] is available, the TCP sender has the information to make intelligent decisions about which packets to retransmit and which packets not to retransmit during Fast Recovery. This document applies to TCP connections that are unable to use the TCP Selective Acknowledgement (SACK) option, either because the option is not locally supported or because the TCP peer did not indicate a willingness to use SACK.

In the absence of SACK, there is little information available to the TCP sender in making retransmission decisions during Fast Recovery. From the three duplicate acknowledgments, the sender infers a packet loss, and retransmits the indicated packet. After this, the data sender could receive additional duplicate acknowledgments, as the data receiver acknowledges additional data packets that were already in flight when the sender entered Fast Retransmit.

In the case of multiple packets dropped from a single window of data, the first new information available to the sender comes when the sender receives an acknowledgment for the retransmitted packet (that is, the packet retransmitted when Fast Retransmit was first entered). If there is a single packet drop and no reordering, then the acknowledgment for this packet will acknowledge all of the packets transmitted before Fast Retransmit was entered. However, if there are multiple packet drops, then the acknowledgment for the retransmitted packet will acknowledge some but not all of the packets transmitted before the Fast Retransmit. We call this acknowledgment a partial acknowledgment.

Along with several other suggestions, [Hoe95] suggested that during Fast Recovery the TCP data sender responds to a partial acknowledgment by inferring that the next in-sequence packet has been lost, and retransmitting that packet. This document describes a modification to the Fast Recovery algorithm in RFC 5681 that incorporates a response to partial acknowledgments received during Fast Recovery. We call this modified Fast Recovery algorithm NewReno, because it is a slight but significant variation of the basic Reno algorithm in RFC 5681. This document does not discuss the other suggestions in [Hoe95] and [Hoe96], such as a change to the ssthresh parameter during Slow-Start, or the proposal to send a new packet for every two duplicate acknowledgments during Fast Recovery. The version of NewReno in this document also draws on other discussions of NewReno in the literature [LM97, Hen98].

We do not claim that the NewReno version of Fast Recovery described here is an optimal modification of Fast Recovery for responding to partial acknowledgments, for TCP connections that are unable to use SACK. Based on our experiences with the NewReno modification in the NS simulator [NS] and with numerous implementations of NewReno, we believe that this modification improves the performance of the Fast Retransmit and Fast Recovery algorithms in a wide variety of scenarios.

2. Terminology and Definitions

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in BCP 14, RFC 2119 [RFC2119]. This RFC indicates requirement levels for compliant TCP implementations implementing the NewReno Fast Retransmit and Fast Recovery algorithms described in this document.

This document assumes that the reader is familiar with the terms SENDER MAXIMUM SEGMENT SIZE (SMSS), CONGESTION WINDOW (cwnd), and FLIGHT SIZE (FlightSize) defined in [RFC5681]. FLIGHT SIZE is defined as in [RFC5681] as follows:

FLIGHT SIZE:
: The amount of data that has been sent but not yet cumulatively acknowledged.

3. The Fast Retransmit and Fast Recovery Algorithms in NewReno

The basic idea of these extensions to the Fast Retransmit and Fast Recovery algorithms described in [RFC5681] is as follows. The TCP sender can infer, from the arrival of duplicate acknowledgments, whether multiple losses in the same window of data have most likely occurred, and avoid taking a retransmit timeout or making multiple congestion window reductions due to such an event.

The standard implementation of the Fast Retransmit and Fast Recovery algorithms is given in [RFC5681]. This section specifies the basic NewReno algorithm. Section 4 describes heuristics for processing duplicate acknowledgments after a retransmission timeout. Sections 5 and 6 provide some guidance to implementors based on experience with NewReno implementations. Several appendices provide more background information and describe variations that an implementor may want to consider when tuning performance for certain network scenarios.

The NewReno modification applies to the Fast Recovery procedure that begins when three duplicate ACKs are received and ends when either a retransmission timeout occurs or an ACK arrives that acknowledges all of the data up to and including the data that was outstanding when the Fast Recovery procedure began.

The NewReno algorithm specified in this document extends the implementation in [RFC5681] by introducing a variable specified as "recover" whose initial value is the initial send sequence number. This new variable is used by the sender to record the send sequence number that must be acknowledged before the Fast Recovery procedure is declared to be over. This variable is used below in step 1, in the response to a partial or new acknowledgment in step 5, and in modifications to step 1 and the addition of step 6 for avoiding multiple Fast Retransmits caused by the retransmission of packets already received by the receiver.

1) Three duplicate ACKs:
When the third duplicate ACK is received and the sender is not already in the Fast Recovery procedure, check to see if the Cumulative Acknowledgment field covers more than "recover". If so, go to Step 1A. Otherwise, go to Step 1B.

1A) Invoking Fast Retransmit:
If so, then set ssthresh to no more than the value given in equation 1 below. (This is equation 4 from [RFC5681]).

```
ssthresh = max (FlightSize / 2, 2*SMSS)           (1)
```

In addition, record the highest sequence number transmitted in the variable "recover", and go to Step 2.

1B) Not invoking Fast Retransmit:
Do not enter the Fast Retransmit and Fast Recovery procedure. In particular, do not change ssthresh, do not go to Step 2 to retransmit the "lost" segment, and do not execute Step 3 upon

subsequent duplicate ACKs.

2) Entering Fast Retransmit:
Retransmit the lost segment and set cwnd to ssthresh plus 3*SMSS. This artificially "inflates" the congestion window by the number of segments (three) that have left the network and the receiver has buffered.

3) Fast Recovery:
For each additional duplicate ACK received while in Fast Recovery, increment cwnd by SMSS. This artificially inflates the congestion window in order to reflect the additional segment that has left the network.

4) Fast Recovery, continued:
Transmit a segment, if allowed by the new value of cwnd and the receiver's advertised window.

5) When an ACK arrives that acknowledges new data, this ACK could be the acknowledgment elicited by the retransmission from step 2, or elicited by a later retransmission.

Full acknowledgments:
If this ACK acknowledges all of the data up to and including "recover", then the ACK acknowledges all the intermediate segments sent between the original transmission of the lost segment and the receipt of the third duplicate ACK. Set cwnd to either (1) min (ssthresh, max(FlightSize, SMSS) + SMSS) or (2) ssthresh, where ssthresh is the value set in step 1; this is termed "deflating" the window. (We note that "FlightSize" in step 1 referred to the amount of data outstanding in step 1, when Fast Recovery was entered, while "FlightSize" in step 5 refers to the amount of data outstanding in step 5, when Fast Recovery is exited.) If the second option is selected, the implementation is encouraged to take measures to avoid a possible burst of data, in case the amount of data outstanding in the network is much less than the new congestion window allows. A simple mechanism is to limit the number of data packets that can be sent in response to a single acknowledgment. Exit the Fast Recovery procedure.

Partial acknowledgments:
If this ACK does *not* acknowledge all of the data up to and including "recover", then this is a partial ACK. In this case, retransmit the first unacknowledged segment. Deflate the congestion window by the amount of new data acknowledged by the cumulative acknowledgment field. If the partial ACK acknowledges at least one SMSS of new data, then add back SMSS bytes to the congestion window. As in Step 3, this artificially

inflates the congestion window in order to reflect the additional segment that has left the network. Send a new segment if permitted by the new value of cwnd. This "partial window deflation" attempts to ensure that, when Fast Recovery eventually ends, approximately ssthresh amount of data will be outstanding in the network. Do not exit the Fast Recovery procedure (i.e., if any duplicate ACKs subsequently arrive, execute Steps 3 and 4 above).

For the first partial ACK that arrives during Fast Recovery, also reset the retransmit timer. Timer management is discussed in more detail in Section 4.

6) Retransmit timeouts:
After a retransmit timeout, record the highest sequence number transmitted in the variable "recover" and exit the Fast Recovery procedure if applicable.

Step 1 specifies a check that the Cumulative Acknowledgment field covers more than "recover". Because the acknowledgment field contains the sequence number that the sender next expects to receive, the acknowledgment "ack_number" covers more than "recover" when:

```
ack_number - 1 > recover;
```

i.e., at least one byte more of data is acknowledged beyond the highest byte that was outstanding when Fast Retransmit was last entered.

Note that in Step 5, the congestion window is deflated after a partial acknowledgment is received. The congestion window was likely to have been inflated considerably when the partial acknowledgment was received. In addition, depending on the original pattern of packet losses, the partial acknowledgment might acknowledge nearly a window of data. In this case, if the congestion window was not deflated, the data sender might be able to send nearly a window of data back-to-back.

This document does not specify the sender’s response to duplicate ACKs when the Fast Retransmit/Fast Recovery algorithm is not invoked. This is addressed in other documents, such as those describing the Limited Transmit procedure [RFC3042]. This document also does not address issues of adjusting the duplicate acknowledgment threshold, but assumes the threshold specified in the IETF standards; the current standard is RFC 5681, which specifies a threshold of three duplicate acknowledgments.

As a final note, we would observe that in the absence of the SACK option, the data sender is working from limited information. When the issue of recovery from multiple dropped packets from a single window of data is of particular importance, the best alternative would be to use the SACK option.

4. Handling Duplicate Acknowledgments After A Timeout

After each retransmit timeout, the highest sequence number transmitted so far is recorded in the variable "recover". If, after a retransmit timeout, the TCP data sender retransmits three consecutive packets that have already been received by the data receiver, then the TCP data sender will receive three duplicate acknowledgments that do not cover more than "recover". In this case, the duplicate acknowledgments are not an indication of a new instance of congestion. They are simply an indication that the sender has unnecessarily retransmitted at least three packets.

However, when a retransmitted packet is itself dropped, the sender can also receive three duplicate acknowledgments that do not cover more than "recover". In this case, the sender would have been better off if it had initiated Fast Retransmit. For a TCP that implements the algorithm specified in Section 3 of this document, the sender does not infer a packet drop from duplicate acknowledgments in this scenario. As always, the retransmit timer is the backup mechanism for inferring packet loss in this case.

There are several heuristics, based on timestamps or on the amount of advancement of the cumulative acknowledgment field, that allow the sender to distinguish, in some cases, between three duplicate acknowledgments following a retransmitted packet that was dropped, and three duplicate acknowledgments from the unnecessary retransmission of three packets [Gur03, GF04]. The TCP sender MAY use such a heuristic to decide to invoke a Fast Retransmit in some cases, even when the three duplicate acknowledgments do not cover more than "recover".

For example, when three duplicate acknowledgments are caused by the unnecessary retransmission of three packets, this is likely to be accompanied by the cumulative acknowledgment field advancing by at least four segments. Similarly, a heuristic based on timestamps uses the fact that when there is a hole in the sequence space, the timestamp echoed in the duplicate acknowledgment is the timestamp of the most recent data packet that advanced the cumulative acknowledgment field [RFC1323]. If timestamps are used, and the sender stores the timestamp of the last acknowledged segment, then the timestamp echoed by duplicate acknowledgments can be used to distinguish between a retransmitted packet that was dropped and three duplicate acknowledgments from the unnecessary retransmission of three packets.

4.1. ACK Heuristic

If the ACK-based heuristic is used, then following the advancement of the cumulative acknowledgment field, the sender stores the value of the previous cumulative acknowledgment as prev_highest_ack, and stores the latest cumulative ACK as highest_ack. In addition, the following step is performed if Step 1 in Section 3 fails, before proceeding to Step 1B.

1*) If the Cumulative Acknowledgment field didn't cover more than "recover", check to see if the congestion window is greater than SMSS bytes and the difference between highest_ack and prev_highest_ack is at most 4*SMSS bytes. If true, duplicate ACKs indicate a lost segment (proceed to Step 1A in Section 3). Otherwise, duplicate ACKs likely result from unnecessary retransmissions (proceed to Step 1B in Section 3).

The congestion window check serves to protect against fast retransmit immediately after a retransmit timeout.

If several ACKs are lost, the sender can see a jump in the cumulative ACK of more than three segments, and the heuristic can fail. RFC 5681 recommends that a receiver should send duplicate ACKs for every out-of-order data packet, such as a data packet received during Fast Recovery. The ACK heuristic is more likely to fail if the receiver does not follow this advice, because then a smaller number of ACK losses are needed to produce a sufficient jump in the cumulative ACK.

4.2. Timestamp Heuristic

If this heuristic is used, the sender stores the timestamp of the last acknowledged segment. In addition, the second paragraph of step 1 in Section 3 is replaced as follows:

1**) If the Cumulative Acknowledgment field didn't cover more than "recover", check to see if the echoed timestamp in the last non-duplicate acknowledgment equals the stored timestamp. If true, duplicate ACKs indicate a lost segment (proceed to Step 1A in Section 3). Otherwise, duplicate ACKs likely result from unnecessary retransmissions (proceed to Step 1B in Section 3).

The timestamp heuristic works correctly, both when the receiver echoes timestamps as specified by [RFC1323], and by its revision attempts. However, if the receiver arbitrarily echoes timestamps, the heuristic can fail. The heuristic can also fail if a timeout was spurious and returning ACKs are not from retransmitted segments. This can be prevented by detection algorithms such as [RFC3522].

5. Implementation Issues for the Data Receiver

[RFC5681] specifies that "Out-of-order data segments SHOULD be acknowledged immediately, in order to accelerate loss recovery." Neal Cardwell has noted that some data receivers do not send an immediate acknowledgment when they send a partial acknowledgment, but instead wait first for their delayed acknowledgment timer to expire [C98]. As [C98] notes, this severely limits the potential benefit of NewReno by delaying the receipt of the partial acknowledgment at the data sender. Echoing RFC 5681, our recommendation is that the data receiver send an immediate acknowledgment for an out-of-order segment, even when that out-of-order segment fills a hole in the buffer.

6. Implementation Issues for the Data Sender

In Section 3, Step 5 above, it is noted that implementations should take measures to avoid a possible burst of data when leaving Fast Recovery, in case the amount of new data that the sender is eligible to send due to the new value of the congestion window is large. This can arise during NewReno when ACKs are lost or treated as pure window updates, thereby causing the sender to underestimate the number of new segments that can be sent during the recovery procedure. Specifically, bursts can occur when the FlightSize is much less than the new congestion window when exiting from Fast Recovery. One simple mechanism to avoid a burst of data when leaving Fast Recovery is to limit the number of data packets that can be sent in response to a single acknowledgment. (This is known as "maxburst_" in the ns simulator.) Other possible mechanisms for avoiding bursts include rate-based pacing, or setting the slow-start threshold to the resultant congestion window and then resetting the congestion window to FlightSize. A recommendation on the general mechanism to avoid excessively bursty sending patterns is outside the scope of this document.

An implementation may want to use a separate flag to record whether or not it is presently in the Fast Recovery procedure. The use of the value of the duplicate acknowledgment counter for this purpose is not reliable because it can be reset upon window updates and out-of-order acknowledgments.

When updating the Cumulative Acknowledgment field outside of Fast Recovery, the "recover" state variable may also need to be updated in order to continue to permit possible entry into Fast Recovery (Section 3, step 1). This issue arises when an update of the Cumulative Acknowledgment field results in a sequence wraparound that affects the ordering between the Cumulative Acknowledgment field and the "recover" state variable. Entry into Fast Recovery is only possible when the Cumulative Acknowledgment field covers more than the "recover" state variable.

It is important for the sender to respond correctly to duplicate ACKs received when the sender is no longer in Fast Recovery (e.g., because of a Retransmit Timeout). The Limited Transmit procedure [RFC3042] describes possible responses to the first and second duplicate acknowledgments. When three or more duplicate acknowledgments are received, the Cumulative Acknowledgment field doesn't cover more than "recover", and a new Fast Recovery is not invoked, it is important that the sender not execute the Fast Recovery steps (3) and (4) in Section 3. Otherwise, the sender could end up in a chain of spurious timeouts. We mention this only because several NewReno implementations had this bug, including the implementation in the NS simulator.

It has been observed that some TCP implementations enter a slow start or congestion avoidance window updating algorithm immediately after the cwnd is set by the equation found in (Section 3, step 5), even without a new external event generating the cwnd change. Note that after cwnd is set based on the procedure for exiting Fast Recovery (Section 3, step 5), cwnd SHOULD NOT be updated until a further event occurs (e.g., arrival of an ack, or timeout) after this adjustment.

7. Security Considerations

RFC 5681 discusses general security considerations concerning TCP congestion control. This document describes a specific algorithm that conforms with the congestion control requirements of RFC 5681, and so those considerations apply to this algorithm, too. There are no known additional security concerns for this specific algorithm.

8. IANA Considerations

This document has no actions for IANA.

9. Conclusions

This document specifies the NewReno Fast Retransmit and Fast Recovery algorithms for TCP. This NewReno modification to TCP can even be important for TCP implementations that support the SACK option, because the SACK option can only be used for TCP connections when both TCP end-nodes support the SACK option. NewReno performs better than Reno (RFC 5681) in a number of scenarios discussed herein.

A number of options to the basic algorithm presented in Section 3 are also described in appendices to this document. These include the handling of the retransmission timer (Appendix A), the response to partial acknowledgments (Appendix B), and whether or not the sender maintains a state variable called "recover" (Appendix C). Our belief is that the differences between these variants of NewReno are small compared to the differences between Reno and NewReno. That is, the important thing is to implement NewReno instead of Reno, for a TCP connection without SACK; it is less important exactly which of the variants of NewReno is implemented.

10. Acknowledgments

Many thanks to Anil Agarwal, Mark Allman, Armando Caro, Jeffrey Hsu, Vern Paxson, Kacheong Poon, Keyur Shah, and Bernie Volz for detailed feedback on this document or on its precursor, RFC 2582. Jeffrey Hsu provided clarifications on the handling of the recover variable that were applied to RFC 3782 as errata, and now are in Section 8 of this document. Yoshifumi Nishida contributed a modification to the fast recovery algorithm to account for the case in which flightsize is 0 when the TCP sender leaves fast recovery, and the TCP receiver uses delayed acknowledgments. Alexander Zimmermann provided several suggestions to improve the clarity of the document.

11. References

11.1. Normative References

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

[RFC2988] Paxson, V. and M. Allman, "Computing TCP’s Retransmission Timer", RFC 2988, November 2000.

[RFC5681] Allman, M., Paxson, V. and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.

11.2. Informative References

[C98] Cardwell, N., "delayed ACKs for retransmitted packets: ouch!". November 1998, Email to the tcpimpl mailing list, Message-ID "Pine.LNX.4.02A.9811021421340.26785-100000@sake.cs.washington.edu", archived at "http://tcp-impl.lerc.nasa.gov/tcp-impl".

[F98] Floyd, S., Revisions to RFC 2001, "Presentation to the TCPIMPL Working Group", August 1998. URLs "ftp://ftp.ee.lbl.gov/talks/sf-tcpimpl-aug98.ps" and "ftp://ftp.ee.lbl.gov/talks/sf-tcpimpl-aug98.pdf".

[F03] Floyd, S., "Moving NewReno from Experimental to Proposed Standard? Presentation to the TSVWG Working Group", March 2003. URLs "http://www.icir.org/floyd/talks/newreno-Mar03.ps" and "http://www.icir.org/floyd/talks/newreno-Mar03.pdf".

[FF96] Fall, K. and S. Floyd, "Simulation-based Comparisons of Tahoe, Reno and SACK TCP", Computer Communication Review, July 1996. URL "ftp://ftp.ee.lbl.gov/papers/sacks.ps.Z".

[F94] Floyd, S., "TCP and Successive Fast Retransmits", Technical report, October 1994. URL "ftp://ftp.ee.lbl.gov/papers/fastretrans.ps".

[GF04] Gurtov, A. and S. Floyd, "Resolving Acknowledgment Ambiguity in non-SACK TCP", Next Generation Teletraffic and Wired/Wireless Advanced Networking (NEW2AN’04), February 2004. URL "http://www.cs.helsinki.fi/u/gurtov/papers/heuristics.html".

[Gur03] Gurtov, A., "[Tsvwg] resolving the problem of unnecessary fast retransmits in go-back-N", email to the tsvwg mailing list, message ID <3F25B467.9020609@cs.helsinki.fi>, July 28, 2003. URL "http://www1.ietf.org/mail-archive/working-groups/tsvwg/current/msg04334.html".

[Hen98] Henderson, T., Re: NewReno and the 2001 Revision. September 1998. Email to the tcpimpl mailing list, Message ID "Pine.BSI.3.95.980923224136.26134A-100000@raptor.CS.Berkeley.EDU", archived at "http://tcp-impl.lerc.nasa.gov/tcp-impl".

[Hoe95] Hoe, J., "Startup Dynamics of TCP’s Congestion Control and Avoidance Schemes", Master’s Thesis, MIT, 1995.

[Hoe96] Hoe, J., "Improving the Start-up Behavior of a Congestion Control Scheme for TCP", ACM SIGCOMM, August 1996. URL "http://www.acm.org/sigcomm/sigcomm96/program.html".

[LM97] Lin, D. and R. Morris, "Dynamics of Random Early Detection", SIGCOMM 97, September 1997. URL "http://www.acm.org/sigcomm/sigcomm97/program.html".

[NS] The Network Simulator (NS). URL "http://www.isi.edu/nsnam/ns/".

[PF01] Padhye, J. and S. Floyd, "Identifying the TCP Behavior of Web

Servers", June 2001, SIGCOMM 2001.

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

[RFC2582] Floyd, S. and T. Henderson, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 2582, April 1999.

[RFC2883] Floyd, S., J. Mahdavi, M. Mathis, and M. Podolsky, "The Selective Acknowledgment (SACK) Option for TCP, RFC 2883, July 2000.

[RFC3042] Allman, M., Balakrishnan, H. and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit", RFC 3042, January 2001.

[RFC3522] Ludwig, R. and M. Meyer, "The Eifel Detection Algorithm for TCP", RFC 3522, April 2003.

[RFC3782] Floyd, S., T. Henderson, and A. Gurtov, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 3782, April 2004.

Appendix A. Resetting the Retransmit Timer in Response to Partial Acknowledgments

One possible variant to the response to partial acknowledgments specified in Section 3 concerns when to reset the retransmit timer after a partial acknowledgment. The algorithm in Section 3, Step 5, resets the retransmit timer only after the first partial ACK. In this case, if a large number of packets were dropped from a window of data, the TCP data sender's retransmit timer will ultimately expire, and the TCP data sender will invoke Slow-Start. (This is illustrated on page 12 of [F98].) We call this the Impatient variant of NewReno. We note that the Impatient variant in Section 3 doesn't follow the recommended algorithm in RFC 2988 of restarting the retransmit timer after every packet transmission or retransmission (step 5.1 of [RFC2988]).

In contrast, the NewReno simulations in [FF96] illustrate the algorithm described above with the modification that the retransmit timer is reset after each partial acknowledgment. We call this the Slow-but-Steady variant of NewReno. In this case, for a window with a large number of packet drops, the TCP data sender retransmits at most one packet per roundtrip time. (This behavior is illustrated in the New-Reno TCP simulation of Figure 5 in [FF96], and on page 11 of [F98]).

When N packets have been dropped from a window of data for a large value of N, the Slow-but-Steady variant can remain in Fast Recovery for N round-trip times, retransmitting one more dropped packet each

round-trip time; for these scenarios, the Impatient variant gives a faster recovery and better performance. The Impatient variant can be particularly important for TCP connections with large congestion windows.

One can also construct scenarios where the Slow-but-Steady variant gives better performance than the Impatient variant. As an example, this occurs when only a small number of packets are dropped, the RTO is sufficiently small that the retransmit timer expires, and performance would have been better without a retransmit timeout.

The Slow-but-Steady variant can also achieve higher goodput than the Impatient variant, by avoiding unnecessary retransmissions. This could be of special interest for cellular links, where every transmission costs battery power and money. The Slow-but-Steady variant can also be more robust to delay variation in the network, where a delay spike might force the Impatient variant into a timeout and go-back-N recovery.

Neither of the two variants discussed above are optimal. Our recommendation is for the Impatient variant, as specified in Section 3 of this document, because of the poor performance of the Slow-but-Steady variant for TCP connections with large congestion windows.

One possibility for a more optimal algorithm would be one that recovered from multiple packet drops as quickly as does slow-start, while resetting the retransmit timers after each partial acknowledgment, as described in the section below. We note, however, that there is a limitation to the potential performance in this case in the absence of the SACK option.

Appendix B. Retransmissions after a Partial Acknowledgment

One possible variant to the response to partial acknowledgments specified in Section 3 would be to retransmit more than one packet after each partial acknowledgment, and to reset the retransmit timer after each retransmission. The algorithm specified in Section 3 retransmits a single packet after each partial acknowledgment. This is the most conservative alternative, in that it is the least likely to result in an unnecessarily-retransmitted packet. A variant that would recover faster from a window with many packet drops would be to effectively Slow-Start, retransmitting two packets after each partial acknowledgment. Such an approach would take less than N roundtrip times to recover from N losses [Hoe96]. However, in the absence of SACK, recovering as quickly as slow-start introduces the likelihood of unnecessarily retransmitting packets, and this could significantly complicate the recovery mechanisms.

We note that the response to partial acknowledgments specified in
Section 3 of this document and in RFC 2582 differs from the respons
e
in [FF96], even though both approaches only retransmit one packet i
n
response to a partial acknowledgment. Step 5 of Section 3 specifie
s
that the TCP sender responds to a partial ACK by deflating the
congestion window by the amount of new data acknowledged, adding
back SMSS bytes if the partial ACK acknowledges at least SMSS bytes
of new data, and sending a new segment if permitted by the new valu
e
of cwnd. Thus, only one previously-sent packet is retransmitted in
response to each partial acknowledgment, but additional new packets
might be transmitted as well, depending on the amount of new data
acknowledged by the partial acknowledgment. In contrast, the
variant of NewReno illustrated in [FF96] simply set the congestion
window to ssthresh when a partial acknowledgment was received. The
approach in [FF96] is more conservative, and does not attempt to
accurately track the actual number of outstanding packets after a
partial acknowledgment is received. While either of these
approaches gives acceptable performance, the variant specified in
Section 3 recovers more smoothly when multiple packets are dropped
from a window of data.

Appendix C. Avoiding Multiple Fast Retransmits

This appendix describes the motivation for the sender's state
variable "recover".

In the absence of the SACK option or timestamps, a duplicate
acknowledgment carries no information to identify the data packet o
r
packets at the TCP data receiver that triggered that duplicate
acknowledgment. In this case, the TCP data sender is unable to
distinguish between a duplicate acknowledgment that results from a
lost or delayed data packet, and a duplicate acknowledgment that
results from the sender's unnecessary retransmission of a data pack
et
that had already been received at the TCP data receiver. Because o
f
this, with the Retransmit and Fast Recovery algorithms in Reno TCP,
multiple segment losses from a single window of data can sometimes
result in unnecessary multiple Fast Retransmits (and multiple
reductions of the congestion window) [F94].

With the Fast Retransmit and Fast Recovery algorithms in Reno TCP,
the performance problems caused by multiple Fast Retransmits are
relatively minor compared to the potential problems with Tahoe TCP,
which does not implement Fast Recovery. Nevertheless, unnecessary
Fast Retransmits can occur with Reno TCP unless some explicit
mechanism is added to avoid this, such as the use of the "recover"
variable. (This modification is called "bugfix" in [F98], and is
illustrated on pages 7 and 9 of that document. Unnecessary Fast
Retransmits for Reno without "bugfix" is illustrated on page 6 of

[F98].)

Section 3 of [RFC2582] defined a default variant of NewReno TCP that did not use the variable "recover", and did not check if duplicate ACKs cover the variable "recover" before invoking Fast Retransmit. With this default variant from RFC 2582, the problem of multiple Fast Retransmits from a single window of data can occur after a Retransmit Timeout (as in page 8 of [F98]) or in scenarios with reordering. RFC 2582 also defined Careful and Less Careful variants of the NewReno algorithm, and recommended the Careful variant.

The algorithm specified in Section 3 of this document corresponds to the Careful variant of NewReno TCP from RFC 2582, and eliminates the problem of multiple Fast Retransmits. This algorithm uses the variable "recover", whose initial value is the initial send sequence number. After each retransmit timeout, the highest sequence number transmitted so far is recorded in the variable "recover".

Appendix D. Simulations

This section provides pointers to simulation scripts available in the NS simulator that reproduce behavior described above.

In Section 3, a simple mechanism is described to limit the number of data packets that can be sent in response to a single acknowledgment. This is known as "maxburst_" in the NS simulator.

Simulations with NewReno are illustrated with the validation test "tcl/test/test-all-newreno" in the NS simulator. The command "../../ns test-suite-newreno.tcl reno" shows a simulation with Reno TCP, illustrating the data sender’s lack of response to a partial acknowledgment. In contrast, the command "../../ns test-suite-newreno.tcl newreno_B" shows a simulation with the same scenario using the NewReno algorithms described in this paper.

Regarding the handling of duplicate acknowledgments after a timeout, the congestion window check serves to protect against fast retransmit immediately after a retransmit timeout, similar to the "exitFastRetrans_" variable in NS. Examples of applying the ACK heuristic (Section 4) are in validation tests "./test-all-newreno newreno_rto_loss_ack" and "./test-all-newreno newreno_rto_dup_ack" in directory "tcl/test" of the NS simulator.
If several ACKs are lost, the sender can see a jump in the cumulative ACK of more than three segments, and the heuristic can fail. A validation test for this scenario is "./test-all-newreno newreno_rto_loss_ackf".

Examples of applying the timestamp heuristic (Section 4) are in

validation tests "./test-all-newreno newreno_rto_loss_tsh" and "./test-all-newreno newreno_rto_dup_tsh".

Section 6 described a problem involving possible spurious timeouts, and mentions that this bug existed in the NS simulator. This bug in the NS simulator was fixed in July 2003, with the variable "exitFastRetrans_".

Regarding the Slow-but-Steady and Impatient variants described in Appendix A, The tests "ns test-suite-newreno.tcl impatient1" and "ns test-suite-newreno.tcl slow1" in the NS simulator illustrate a scenario in which the Impatient variant performs better than the Slow-but-Steady variant. The Impatient variant can be particularly important for T
CP
connections with large congestion windows, as illustrated by the te
sts
"ns test-suite-newreno.tcl impatient4" and "ns test-suite-newreno.t
cl
slow4" in the NS simulator. The tests "ns test-suite-newreno.tcl impatient2" and "ns test-suite-newreno.tcl slow2" in the NS simulator illustrate scenarios in which the Slow-but-Steady variant outperforms the Impa
tient
variant. The tests "ns test-suite-newreno.tcl impatient3" and "ns test-suite-newreno.tcl slow3" in the NS simulator illustrate scenarios in which the Slow-but-Steady variants avoid unnecessary retransmissions.

Appendix B describes different policies for partial window deflatio
n.
The [FF96] behavior can be seen in the NS simulator by setting the variable "partial_window_deflation_" for "Agent/TCP/Newreno" to 0; the behavior specified in Section 3 is achieved by setting "partial_window_deflation_" to 1.

Section 3 of [RFC2582] defined a default variant of NewReno TCP tha
t
did not use the variable "recover", and did not check if duplicate ACKs cover the variable "recover" before invoking Fast Retransmit. With this default variant from RFC 2582, the problem of multiple Fa
st
Retransmits from a single window of data can occur after a Retransm
it
Timeout (as in page 8 of [F98]) or in scenarios with reordering (as An NS validation test "./test-all-newreno newreno5_noBF" in directory "tcl/test" of the NS simulator illustartes the default variant of NewReno TCP that doesn’t use the variable "recover"; this gives performance similar to that on page 8 of [F03].

Appendix E. Comparisons between Reno and NewReno TCP

As we stated in the introduction, we believe that the NewReno modification described in this document improves the performance of the Fast Retransmit and Fast Recovery algorithms of Reno TCP in a wide variety of scenarios. This has been discussed in some depth i
n

[FF96], which illustrates Reno TCP's poor performance when multiple packets are dropped from a window of data and also illustrates NewReno TCP's good performance in that scenario.

We do, however, know of one scenario where Reno TCP gives better performance than NewReno TCP, that we describe here for the sake of completeness. Consider a scenario with no packet loss, but with sufficient reordering so that the TCP sender receives three duplicate acknowledgments. This will trigger the Fast Retransmit and Fast Recovery algorithms. With Reno TCP or with Sack TCP, this will result in the unnecessary retransmission of a single packet, combined with a halving of the congestion window (shown on pages 4 and 6 of [F03]). With NewReno TCP, however, this reordering will also result in the unnecessary retransmission of an entire window of data (shown on page 5 of [F03]).

While Reno TCP performs better than NewReno TCP in the presence of reordering, NewReno's superior performance in the presence of multiple packet drops generally outweighs its less optimal performance in the presence of reordering. (Sack TCP is the preferred solution, with good performance in both scenarios.) This document recommends the Fast Retransmit and Fast Recovery algorithms of NewReno TCP instead of those of Reno TCP for those TCP connections that do not support SACK. We would also note that NewReno's Fast Retransmit and Fast Recovery mechanisms are widely deployed in TCP implementations in the Internet today, as documented in [PF01]. For example, tests of TCP implementations in several thousand web servers in 2001 showed that for those TCP connections where the web browser was not SACK-capable, more web servers used the Fast Retransmit and Fast Recovery algorithms of NewReno than those of Reno or Tahoe TCP [PF01].

Appendix F. Changes Relative to RFC 2582

The purpose of this document is to advance the NewReno's Fast Retransmit and Fast Recovery algorithms in RFC 2582 to Standards Track.

The main change in this document relative to RFC 2582 is to specify the Careful variant of NewReno's Fast Retransmit and Fast Recovery algorithms. The base algorithm described in RFC 2582 did not attempt to avoid unnecessary multiple Fast Retransmits that can occur after a timeout (described in more detail in the section above). However, RFC 2582 also defined "Careful" and "Less Careful" variants that avoid these unnecessary Fast Retransmits, and recommended the Careful variant. This document specifies the previously-named "Careful" variant as the basic version of NewReno. As described below, this algorithm uses a variable "recover", whose initial value is the send sequence number.

The algorithm specified in Section 3 checks whether the acknowledgment field of a partial acknowledgment covers *more* than "recover", as defined in Section 3. Another possible variant would be to simply require that the acknowledgment field covers *more than or equal to* "recover" before initiating another Fast Retransmit. We called this the Less Careful variant in RFC 2582.

There are two separate scenarios in which the TCP sender could receive three duplicate acknowledgments acknowledging "recover" but no more than "recover". One scenario would be that the data sender transmitted four packets with sequence numbers higher than "recover", that the first packet was dropped in the network, and the following three packets triggered three duplicate acknowledgments acknowledging "recover". The second scenario would be that the sender unnecessarily retransmitted three packets below "recover", and that these three packets triggered three duplicate acknowledgments acknowledging "recover". In the absence of SACK, the TCP sender is unable to distinguish between these two scenarios.

For the Careful variant of Fast Retransmit, the data sender would have to wait for a retransmit timeout in the first scenario, but would not have an unnecessary Fast Retransmit in the second scenario. For the Less Careful variant to Fast Retransmit, the data sender would Fast Retransmit as desired in the first scenario, and would unnecessarily Fast Retransmit in the second scenario. This document only specifies the Careful variant in Section 3. Unnecessary Fast Retransmits with the Less Careful variant in scenarios with reordering are illustrated in page 8 of [F03].

The document also specifies two heuristics that the TCP sender MAY use to decide to invoke Fast Retransmit even when the three duplicate acknowledgments do not cover more than "recover". These heuristics, an ACK-based heuristic and a timestamp heuristic, are described in Sections 6.1 and 6.2 respectively.

Appendix G. Changes Relative to RFC 3782

In [RFC3782], the cwnd after Full ACK reception will be set to (1) min (ssthresh, FlightSize + SMSS) or (2) ssthresh. However, there is a risk in the first logic which results in performance degradation. With the first logic, if FlightSize is zero, the result will be 1 SMSS. This means TCP can transmit only 1 segment at this moment, which can cause delay in ACK transmission at receiver due to delayed ACK algorithm.

The FlightSize on Full ACK reception can be zero in some situations. A typical example is where sending window size during fast recovery is small. In this case, the retransmitted packet and new data packets can

be transmitted within a short interval. If all these packets successfully arrive, the receiver may generate a Full ACK that acknowledges all outstanding data. Even if window size is not smal
l,
loss of ACK packets or receive buffer shortage during fast recovery
can
also increase the possibility to fall into this situation.

The proposed fix in this document ensures that sender TCP transmits
at
least two segments on Full ACK reception.

In addition, errata for RFC3782 (editorial clarification to Section
8
of RFC2582, which is now Section 6 of this document) has been appli
ed.

Sections 4, 5, and 9-11 of RFC2582 were relocated to appendices of this document since they are non-normative and provide background information and references to simulation results.

Appendix H. Document Revision History

To be removed upon publication

Revision	Comments
draft-00	RFC3782 errata applied, and changes applied from draft-nishida-newreno-modification-02
draft-01	Non-normative sections moved to appendices, editorial clarifications applied as suggested by Alexander Zimmermann.

Authors' Addresses

Tom Henderson
The Boeing Company

EMail: thomas.r.henderson@boeing.com

Sally Floyd
International Computer Science Institute

Phone: +1 (510) 666-2989
EMail: floyd@acm.org
URL: http://www.icir.org/floyd/

Andrei Gurtov
HIIT
Helsinki Institute for Information Technology
P.O. Box 19215
00076 Aalto
Finland

EMail: gurtov@hiit.fi

Yoshifumi Nishida
WIDE Project
Endo 5322
Fujisawa, Kanagawa 252-8520
Japan

Email: nishida@wide.ad.jp

This Internet-Draft, draft-ietf-tcpm-tcp-security-01.txt, has expired, and has b
een deleted
from the Internet-Drafts directory. An Internet-Draft expires 185 days from
the date that it is posted unless it is replaced by an updated version, or the
Secretariat has been notified that the document is under official review by the
IESG or has been passed to the RFC Editor for review and/or publication as an
RFC. This Internet-Draft was not published as an RFC.

Internet-Drafts are not archival documents, and copies of Internet-Drafts that h
ave
been deleted from the directory are not available. The Secretariat does not hav
e
any information regarding the future plans of the author(s) or working group, if
applicable, with respect to this deleted Internet-Draft. For more information,
or
to request a copy of the document, please contact the author(s) directly.

Draft Author(s):
Fernando Gont <fernando@gont.com.ar>

TCP Maintenance Working Group
Internet-Draft
Intended status: Experimental
Expires: September 8, 2011

M. Mathis
N. Dukkipati
Y. Cheng
Google, Inc
March 7, 2011

Proportional Rate Reduction for TCP
draft-mathis-tcpm-proportional-rate-reduction-00.txt

Abstract

This document describes a pair experimental algorithms, Proportional Rate Reduction (PPR) and Reduction Bound (RB) that improve the accuracy of the amount of data sent by TCP during loss recovery. Standard Congestion Control requires that TCP and other protocols reduce their congestion window in response to losses. This window reduction naturally occurs in the same round trip as the data retransmissions to repair the losses, and is implemented by choosing not to transmit any data in response to some ACKs arriving from the receiver. There are two widely deployed algorithms used to implement this window reduction: Fast Recovery and Rate Halving. Both algorithms are needlessly fragile under a number of conditions, particularly when there is a burst of losses that such that the number of ACKs delivered is so small that the effective window falls below ssthresh, the target value chosen by the congestion control algorithm. Proportional Rate Reduction avoids these excess window reductions such that at the end of recovery the actual window size will be as close as possible to the window size determined by the congestion control algorithm. It is patterned after rate halving, but using the fraction that is appropriate for target window chosen by the congestion control algorithm. In addition a second algorithm, Reduction Bound, monitors the total window reduction due to all mechanisms, including application stalls, the losses themselves and inhibits further window reductions when possible.

Status of this Memo

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 September 8, 2011.

Copyright Notice

Table of Contents

1. Introduction

This document describes a pair experimental algorithms, Proportional Rate Reduction (PPR) and Reduction Bound (RB) that improve the accuracy of the amount of data sent by TCP during loss recovery.

Standard Congestion Control [RFC 5681] requires that TCP (and other protocols) reduce their congestion window in response to losses. Fast Recovery, described in the same document, is the reference algorithm for making this adjustment. It's stated goal is to recover TCP's self clock by relying on returning ACKs during recovery to clock more data into the network. Fast Recovery adjusts the window by waiting for one half RTT of ACKs to pass before sending any data. It is fragile because it can not compensate for the implicit window reduction caused by the losses them selves, and is exposed to timeouts. For example if half of the data or ACKs are lost, Fast Recovery's expected behavior would be to reduce the window by not sending in response to the first half window of ACKs, but then it would not receive any more ACKs and would timeout because it failed to send anything at all.

The rate-halving algorithm improves this situation by sending data on alternate ACKs during recovery, such that after one RTT the window has been halved. Rate-having is implemented in Linux, after being only informally published[RHweb] including an uncompleted Internet-Draft[RHID]. Rate-halving also does not adequately compensate for the implicit window reduction caused by the losses and also assumes a 50% window reduction, which was completely standard at the time it was written. (Several modern congestion control algorithms, such as Cubic[CUBIC], can sometimes reduce the window by much less than 50%.) As a consequence rate-halving often allows the window to fall further than necessary, reducing performance and increasing the risk of timeouts if there are any additional losses.

Proportional Rate Reduction (PPR) avoids these excess window reductions such that at the end of recovery the actual window size will be as close as possible to the window size determined by the congestion control algorithm. It is patterned after Rate Halving, but using the fraction that is appropriate for target window chosen by the congestion control algorithm. In addition, a second algorithm, Reduction Bound (RB), monitors the total window reduction due to all mechanisms, including application stalls, the losses themselves and attempts to inhibit further window reductions.

The foundation of Proportional Rate Reduction is Van Jacobson's packet conservation principle: segments delivered to the receiver are used as the clock to trigger sending additional segments into the network. As much as possible Proportional Rate Reduction and

Reduction Bound rely on this self clock process, and are only slightly affected by the accuracy of other estimators, such as pipe[RFC 3517] and cwnd. This is what gives the algorithms their precision in the presence of events that cause uncertainty in other estimators.

Note that in the round trip time following the detection of a loss TCP has to balance three partially conflicting actions: retransmitting the missing data needed to repair the losses, sending as much new data as possible to preserve TCP's self clock, and not sending data in response to some of the ACKs in order to make the window adjustment prescribed by the congestion control algorithm. We use the term "Voluntary Window Reduction", to refer to this last process: choosing not to send data in response to an ACK that would otherwise permit it.

These algorithms are described as modifications to RFC 5681, TCP Congestion Control, using concepts drawn from the pipe algorithm [RFC 3517]. They are most accurate and more easily implemented with SACK[RFC 2018], but they can be implemented without SACK.

2. Definitions

The following terms, parameters and state variables are used as they are defined in earlier documents:

RFC 3517: covered

RFC 5681: duplicate ACK, FlightSize, Receiver Maximum Segment Size (RMSS)

We define some additional variables:

SACKd: The total number of bytes that the scoreboard indicates has been delivered to the receiver. This can be computed by scanning the scoreboard and counting the total number of bytes covered by all sack blocks.

DeliveredData: The total number of bytes that the current ACK indicates have been delivered to the receiver, relative to all past ACKs. When not in recovery, DeliveredData is the change in snd.una. With SACK, DeliveredData is not an estimator and can be computed precisely as the change in snd.una plus the change in SACKd. Note that if there are SACK blocks and snd.una advances, the change in SACKd is typically negative. In recovery without SACK, DeliveredData is estimated to be 1 rmss on duplicate acknowledgements, and on a subsequent partial or full ACK, DeliveredData is estimated to be the change in snd.una, minus one rmss for each preceding duplicate ACK.

Note that DeliveredData is robust: for TCP using SACK, DeliveredData can be precisely computed anywhere in the network just by inspecting the returning ACKs. The consequence of missing ACKs is that later ACKs will show a larger DeliveredData, and that for any TCP the sum of DeliveredData must agree with the forward progress over the same time interval.

We introduce a local variable "sndcnt", which indicates exactly how many bytes should be sent in response to each ACK while in recovery. Note that the decision of which data to send (e.g. retransmit missing data or send more new data) is out of scope for this document.

3. Algorithm

At the beginning of recovery initialize state. This assumes a modern congestion control algorithm, CongCtrlAlg(), that might set ssthresh to something other than FlightSize/2:

```
ssthresh = CongCtrlAlg() // Target cwnd after recovery
prr_delivered = 0         // Total bytes delivered during recov
prr_out = 0              // Total bytes sent during recovery
RecoverFS = snd.nxt-snd.una // Flightsize at the start of recov
pipe = as defined in [RFC 3517] // Estimated bytes in the network
```

On every ACK that advances snd.una compute:

```
DeliveredData = delta(snd.una) + delta(SACKd)
prr_delivered += DeliveredData
pipe = (RFC 3517 pipe algorithm)
if (pipe > ssthresh) {
   // Proportional Rate Reduction
   sndcnt = CEIL(prr_delivered * ssthresh / RecoverFS) - prr_out
} else {
   // Reduction Bound
   sndcnt = MIN(ssthresh - pipe, prr_delivered - prr_out)
}
sndcnt = MAX(sndcnt, 0)                    // positive
```

On any data transmission or retransmission:

```
prr_out += (data sent) // strictly less than or equal to sndcnt
```

Algorithm summary: If pipe (the estimated data is in flight) is larger than ssthresh (the target cwnd at the end of recovery) then Proportional Rate Reduction spreads the the voluntary window reductions across a full RTT, such that at the end of recovery (as prr_delivered approaches RecoverFS) prr_out approaches ssthresh, the target value for cwnd. If there are excess losses such that pipe falls below ssthresh, Reduction Bound first tries to hold pipe at ssthresh by undoing past voluntary window reductions (as long as prr_delivered > prr_out). While there are past voluntary window reductions single recovery ACKs can trigger sending multiple segments. If there are too many losses then prr_delivered - prr_out will be exactly the same as DeliveredData for the current ACK, resulting in sndcnt = DeliveredData and there will be no further Voluntary Window Reductions.

4. Algorithm Properties

Normally Proportional Rate Reduction will spread Voluntary Window reductions out evenly across a full RTT. This has the potential to generally reduce the burstiness of Internet traffic, and could be considered to be a type of soft pacing. Theoretically any pacing increases the probability that different flows are interleaved, reducing the opportunity for ACK compression and other phenomena that increase traffic burstiness. However these effects have not been quantified.

If there are minimal losses, Proportional Rate Reduction will converge to exactly the target window chosen by the congestion control algorithm. Note that as TCP approaches the end of recovery prr_delivered will approach RecoverFS and sndcnt will be computed such that prr_out approaches ssthresh.

Implicit window reductions due to multiple isolated losses during recovery cause later Voluntary Reductions to be skipped. For small numbers of losses the window size ends at exactly the window chosen by the congestion control algorithm.

For burst losses, earlier Voluntary Window Reductions can be undone by sending extra segments in response to ACKs arriving later during recovery. Note that as long as some Voluntary Window Reductions are not undone, the final value for pipe will be the same as ssthresh, the target cwnd value chosen by the congestion control algorithm.

At every ACK, cumulative data sent during recovery is strictly bound by the cumulative data delivered to the receiver during recovery. This property is referred to as the "Relentless bound", because it parallels the congestion control algorithm used in Relentless TCP[Relentless]. Any smaller bound implies that we unnecessarily gave up a opportunity to transmit data, and any larger bound has pathological behavior in some network topologies. See Section

Section 6 for a further discussion of this property.

Proportional Rate Reduction with Reduction Bound improves the situation when there are application stalls (e.g. when the sending application does not queue data for transmission quickly enough or the receiver stops advancing rwnd). When there is a application stall early during recovery prr_out will fall behind the sum of the transmissions permitted by sndcnt. The missed opportunities to send due to stalls are treated like banked Voluntary Window Reductions: specifically they cause prr_delivered-prr_out to be significantly positive. If the application catches up while TCP is still in recovery, TCP will send a partial window burst to catch up to exactly where it would have been, had the application never stalled. Although this burst might be viewed as being hard on the network, this is exactly what happens every time there is a partial RTT application stall while not in recovery. We have made the partial RTT stall behavior uniform in all states. Improving this behavior is out of scope for this document.

Proportional Rate Reduction with Reduction Bound is significantly less sensitive to errors of the pipe estimator. While in recovery, pipe is intrinsically an estimator, using incomplete information to guess if un-SACKed segments are actually lost or out-of-order in the network. Under some conditions pipe can have significant errors, for example when a burst of reordered data is presumed to be lost and is retransmitted, but then the original data arrives before the retransmission. If the transmissions are regulated directly by pipe as they are in RFC 3517, then errors and discontinuities in the pipe estimator can cause significant errors in the amount of data sent. With Proportional Rate Reduction with Reduction Bound, pipe merely determines how sndcnt is computed from DataDelivered. Since short term errors in pipe are smoothed out across multiple ACKs and both Proportional Rate Reduction and Reduction Bound converge to the same final window, errors in the pipe estimator have less impact on the final outcome (This needs to be tested better).

5. Comparison to Fast Recovery and other algorithms

To compare PRR-RB to other recovery algorithms, consider how the voluntary window reductions are distributed during TCP recovery. With PRR they are spread evenly across the recovery RTT, such that the final window is determined by the congestion control algorithm.

With Fast Recovery, the voluntary window reductions all occur during the first half of the recovery RTT, before TCP has a sufficient measure of the total lost data or ACKs. The possibility exists that TCP will only receive half of the expected number of ACKs, and will "voluntarily" reduce the window to zero, causing a timeout. Fast Recovery does more quickly free space at a bottleneck network queue, because the voluntary window reductions happen on average a quarter of an RTT earlier than PRR or Ratehalving. It is unknown if this has any significant effect on overall Internet traffic dynamics.

Rate halving also schedules the voluntary window reductions on alternate ACKs, but with insufficient attention to how low the window has fallen.

An alternative algorithm could transmit one segment in response to every segment delivered to the receiver (the relentless bound, see below) until prr_out reaches sshtresh, and then stop transmitting entirely until there is a full or partial ACK. Although this approach minimizes the chances of the actual window falling too low, it is likely to reduce the robustness of the data retransmission and recovery strategy, because algorithms to detect lost retransmissions require sending new data following retransmissions[CITE?].

An even more aggressive algorithm could follow the relentless bound all the way to the end of recovery, and then make the window adjustment after the end of recovery. While this is the absolutely maximally aggressive recovery strategy (see the next section), it has the potential to be unfair, because delaying the window adjustment by one RTT will have an adverse effect on other flows sharing the link.

[Add Concluding Remarks]

6. Packet Conservation Bound

Under all conditions and sequences of events during recovery, PRR-RB strictly bounds the data transmitted to be equal to or less than the amount of data delivered to the receiver. We claim that this packet conservation bound is the most aggressive algorithm that does not lead to pathological behaviors (additional forced losses) in some environments. Furthermore, any less aggressive bound will result in missed opportunities to safely send data without inordinate risk of loss. While we believe that this assertion might be formally provable, we demonstrate it with a little thought experiment:

Imagine a network path that has insignificant delays in both directions, except the processing time and queue at a single bottleneck in the forward path. By insignificant delay, I mean when a packet is "served" at the head of the bottleneck queue, the following events happen in much less than one packet time at the bottleneck: the packet arrives at the receiver; the receiver sends an ACK; which arrives at the sender; the sender processes the ACK and sends some data; the data is queued at the bottleneck.

If sndcnt is set to DataDelivered and nothing else is inhibiting sending data, then clearly the data arriving at the bottleneck queue will exactly replace the data that was served at the head of the queue, so the queue will have a constant length. If queue is drop tail and full then the queue will stay exactly full, even in the presence of losses or reordering on the ACK path, and independent of whether the data is in order or out-of-order (e.g. simple reordering or loss recovery from an earlier RTT). Any more aggressive algorithm, sending additional data will cause a queue overflow and loss. Any less aggressive algorithm will under fill the queue. Therefore setting sndcnt to DataDeliverd is the most aggressive algorithm that does not cause forced losses in this simple network. Relaxing the assumptions (e.g. making delays more authentic and adding more flows, delayed ACKs, etc) increases the noise (jitter) in the system but does not change it's basic behavior.

Note that the congestion control algorithm implements a broader notion of optimal that includes appropriately sharing of the network. PRR-RB will normally choose to send less data than permitted by this bound as it brings the TCP's actual window down to ssthresh, as chosen by the congestion control algorithm.

7. Acknowledgements

This draft is based in part on previous incomplete work by Matt Mathis, Jeff Semke and Jamshid Mahdavi[RHID] and influenced by several discussion with John Heffner.

8. Security Considerations

Proportional Rate Reduction does not change the risk profile for TCP.

Implementers that change PRR from counting bytes to segments have to be cautious about the effects of ACK splitting attacks[SPLIT], where the receiver acknowledges partial segments for the purpose of confusing the sender's congestion accounting.

9. IANA Considerations

This document makes no request of IANA.

Note to RFC Editor: this section may be removed on publication as an RFC.

Appendix A. References

TODO: A proper reference section.

[RFC 3517] "A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP". E. Blanton, M. Allman, K. Fall, L. Wang. April 2003.

[RFC 5681] "TCP Congestion Control". M. Allman, V. Paxson, E. Blanton. September 2009.

[RHweb] "TCP Rate-Halving with Bounding Parameters". M. Mathis, J. Madavi, http://www.psc.edu/networking/papers/FACKnotes/971219/, Dec 1997.

[RHID] "The Rate-Halving Algorithm for TCP Congestion Control". M. Mathis, J. Semke, J. Mahdavi, K. Lahey. http://www.psc.edu/networking/ftp/papers/draft-ratehalving.txt, Work in progress, last updated June 1999.

[CUBIC] "CUBIC: A new TCP-friendly high-speed TCP variant". I. Rhee, L. Xu, PFLDnet, Feb 2005.

Authors' Addresses

Matt Mathis
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 93117
USA

Email: mattmathis@google.com

Nandita Dukkipati
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 93117
USA

Email: nanditad@google.com

Yuchung Cheng
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 93117
USA

Email: ycheng@google.com

TCP Maintenance and Minor Extensions (tcpm)
Internet-Draft
Intended status: Experimental
Expires: September 15, 2011

R. Scheffenegger, Ed.
NetApp, Inc.
M. Kuehlewind
University of Stuttgart
March 14, 2011

Additional negotiation in the TCP Timestamp Option field during the TCP handshake
draft-scheffenegger-tcpm-timestamp-negotiation-01

Abstract

RFC 1323 defines the TSecr field of a SYN packet to be not valid and thus this field will always be zero. This documents specifies the use of this field to signal and negotiate additional information about the content of the TSopt field as well as the behavior of the receiver. If the receiver understands this extension, it will use the TSecr field of the SYN/ACK to reply. Otherwise the receiver will ignore the TSecr field and set a timestamp in the TSecr field as specified in RFC 1323.

Table of Contents

1. Introduction

The TCP Timestamps Option (TSopt) provides timestamp echoing for Round-trip Time (RTT) measurments. TSopt is widely deployed and activated by default in many systems. RFC 1323 [RFC1323] specifies TSopt the following way:

Kind: 8

Length: 10 bytes

```
+-------+-------+---------------------+---------------------+
|Kind=8 |  10   |   TS Value (TSval)  |TS Echo Reply (TSecr)|
+-------+-------+---------------------+---------------------+
    1       1              4                     4
```

RFC1323 TSopt

"The Timestamps option carries two four-byte timestamp fields. The Timestamp Value field (TSval) contains the current value of the timestamp clock of the TCP sending the option.

The Timestamp Echo Reply field (TSecr) is only valid if the ACK bit is set in the TCP header; if it is valid, it echos a timestamp value that was sent by the remote TCP in the TSval field of a Timestamps option. When TSecr is not valid, its value must be zero. The TSecr value will generally be from the most recent Timestamp option that was received; however, there are exceptions that are explained below.

A TCP may send the Timestamps option (TSopt) in an initial SYN segment (i.e., segment containing a SYN bit and no ACK bit), and may send a TSopt in other segments only if it received a TSopt in the initial SYN segment for the connection."

The comparison of the timestamp in the TSecr field to the current time gives an estimation of the RTT. RFC 1323 [RFC1323] specifies various cases when more than one timestamp is available to echo. The proposed solution might not always be the best choice, e.g. when the TCP Selective Acknowledgment Option (SACK) is used. Moreover, more and more use cases arise where one-way delay (OWD) measurements are needed. These mechanism misuse usually the TSopt to estimated the variation in OWD. To enable such mechanisms the TSecr field in the TCP SYN packet could be used for additional negotiation.

1.1. 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 [RFC2119].

2. Overview

Enhancements in the area of TCP congestion control can use the measurement of the one-way delay variance as one input. However, without explicit knowledge of the partner’s timestamp clock, arriving at a good estimate requires multiple segent exchanges over a few round trip times. Nevertheless, any such calculation has to make assumptions about the network state at the time of the measurement. In order to assist such algorithms, explicit knowledge at a early phase of the session can be negotiated.

In addition, by using synergistic signalling between Timestamps and other options such as selective acknowledgment, enhancments in loss recovery are possible by removing any retransmission ambiguity . However, currently receivers are required to only reflect the timestamp of the last segment that was received in order. Therefore, a backwards compatible way of changing this behavior is required.

Furthermore, as the importance of the timestamp option increases by using it in more aspects of a TCP senders algorithm, so increases the importance of maintaining the integrity of the reflected timestamps, while allowing the receiver to make use of a senders timestamp.

As an optional extension, a timestamp clock rate range negotiation is also introduced. However, this is only included as example of further possible enhancements.

3. Definitions

The reader is expected to be familiar with the definitions given in [RFC1323].

4. Signaling

During the initial TCP three-way handshake, timestamp options are negotiated using the TSecr field. A compliant TCP receiver will XOR the flags with the received TSval, when responding with the SYN+ACK. Timestamp Options MAY only be present when the SYN bit is set.

4.1. Capability Flags

In order to signal the supported capabilities, the TSecr is overloaded with the following flags and fields during the three-way handshake. If optional capabilities such as tcp clock range are presented, minimal state will be required in the host to decode the returned Flags xor’ed with the TSval.

Kind: 8

Length: 10 bytes

```
        +-------+-------+---------------------+---------------------+
        |Kind=8 |  10   |   TS Value (TSval)  |TS Echo Reply (TSecr)|
        +-------+-------+---------------------+---------------------+
            1       1              4          |           4         |
                                             /                      |
        .-----------------------------------'                       |
       /                                                             \
      |                                                               |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |E|R|R|         |               |S|         |                   |
      |X|E|E|  MASK   |      RES      |G|  EXP16  |      FRAC16       |
      |O|S|S|         |               |N|         |                   |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

timestamp option flags

EXO - Extended Options
Indicated that the sender supports extended timestamp options as defined by this document, and MUST be set ("1") by compliant implementations.

RES - Reserved
Reserved for future use. MUST not be set ("0"). If a timestamp option is received with this bit set, the receiver MUST ignore the extended options field and react as if the Flags were not set (compatibility mode).

MASK - Mask Timestamps
If the timestamp is used for congestion control purposes, an incentive exists for malicious receivers to reflect tampered timestamps. A sender MAY choose to protect timestamps from such modifications by including a fingerprint (secure hash of some kind) in some of the least significant bits. However, doing so would prevent a receiver from using the timestamp for other purposes. The MASK field indicates how many least significant bits should be excluded by the receiver, when processing a timestamp for timing purposes. Note that this does not impact the reflected timestamp in any way - TSecr will always be equal to an appropriate TSval. Another use case would be when the sender does not support a timestamp clock which can guarantee unique timestamps for retransmitted segments. For unambigously identifying regular from retransmitted segments, the timestamp must be unique for otherwise identical segments. Reserving the least significant bits for this purpose allows senders with slow

running timestamp clocks to make use of this feature. Note that the use of this option as implications in the protection against wraped sequence numbers (PAWS - [RFC1323]), as the more bits are set aside for tamper prevention, the faster the timestamp number space cycles itself.

SGN - binary16 Sign
This is the sign bit of the IEEE 754-2008 binary16 floating point representation of the timestamp clock. Timestamp clocks MUST be positive, thus this bit MUST be zero.

EXP16 - binary16 Exponent
The exponent component of a binary16 floating point number indicating the timestamp clock. The exponent bias is 28, which is not identical to the binary16 definition in IEEE 754-2008. Subnormal numbers (lower precision), where the exponent is set to zero, extend the lowest possible value representation to 2^-38 (or 7.276 ps) at reduced precision. Infinity and NaN (all exponent bits set) are not suppored, an exponent value of 31 is to be treated as normal exponent. This allows timestamp clock rates of up to 15.999 sec.

FRAC16 - binary16 Fraction
The fraction component of a binary16 floating point number indicating the timestamp clock. The clock rate is measured in seconds between ticks. The range with the highest resolution, excluding subnormal numbers, covers clock ranges between 7.45 ns and 15.99 sec. It is expected, that timestamp clock rates in excess of 0.1 ms are implemented by inserting the timestamp "late" before transmitting a segment.

Example for an extended timestamp option, to indicate that the senders timestamp clock (tcp clock) is running with 1 ms per tick:

SYN, TSopt=<X>, TSecr=EXO|MASK|EXP16=18|FRAC16=0x018

The clock rate calculates as 2^(18-28)*1.0000011b, thus indicates an actual clock rate of 999.45 us

4.2. Implicit extended negotiation

If both timestamp extended options and selective acknowledgement options ([RFC2018]) are negotiated, both hosts MUST mirror the timestamp option immediately after receiving it. Note that this is in conflict with [RFC1323], where only the timestamp of the last segment received in sequence is mirrored. As SACK allows discrimination of reordered or lost segments, the reflected timestamps are not required to convey the most conservative information. If SACK indicates lost or reordered packets at the receiver, the sender MUST take appropriate action such as ignoring the received timestamps for calculating the round trip time, or assuming a delayed packet (with appropriate handling). The exact implications are beyond the scope of this note.

This allows the synergistic use of the timestamp option with the SACK option to improve loss recovery, round trip time and one way delay variance measurements even during loss or reordering episodes. This is enabled by removing any retransmission ambiguity using unique timestamps for every retransmission, while simultaneously the SACK option indicates the ordering of received segments even in the presence of ACK loss or reordering.

5. Discussion

One-way delay (variation) based congestion controls would benefit from knowing the clock resolution on both sides.

RTT variance during loss episodes is not deeply researched. Current heuristics (RFC1122, RFC1323, Karn’s algorithm, RFC2988) explicitly exclude (and prevent) the use of RTT samples when loss occurs. However, solving the retransmission ambiguity problem - and the related reliable ACK delivery problem - may allow the refinement of these algorithms further, as well as enabling new research to distinguish between corruption loss (without RTT / one-way delay impact) and congestion loss (with RTT / one-way delay impact). Research into this field appears to be a rather neglected, especially when it comes to large scale, public internet investigations. Due to the very nature of this, passive investigations without signals contained within the headers are only of limited use in empirical research.

Retransmission ambiguity detection during loss recovery would allow an additional level of loss recovery control without reverting to timer-based methods. As with the deployment of SACK, separating "what" to send from "when" to send it could be driven one step further. In particular, less conservative loss recovery schemes which do not trade principles of packet conservation against timeliness, require a reliable way of prompt and best possible feedback from the receiver about any delivered segment and their ordering. SACK alone goes quite a long way, but using Timestamp information in addition could remove any ambiguity. However, the current specs in RFC1323 make that use impossible, thus a modified signaling (receiver behavior) is a necessity.

6. Acknowledgements

The authors would like to thank Dragana Damjanovic for some initial thoughts around Timestamps and their extended potential use.

7. IANA Considerations

This memo includes no request to IANA.

8. Security Considerations

The algorithm presented in this paper shares security considerations with [RFC1323].

Some implementations address the vulerabilities of [RFC1323], by dedicating a few low-order bits of the timestamp fields for use with a (secure) hash, that protects against malicious tweaking of TSecr values. A Flag-field has been provided to transparently notify the receiver about that use of low-order bits, so that they can be excluded in one-way delay calculations.

9. References

9.1. Normative References

[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.

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

9.2. Informative References

[Chirp] Kuehlewind, M. and B. Briscoe, "Chirping for Congestion Control - Implementation Feasibility", Nov 2010, <http://bobbriscoe.net/projects/netsvc_i-f/chirp_pfldnet10.pdf>.

[I-D.ietf-tcpm-tcp-security] Gont, F., "Security Assessment of the Transmission Control Protocol (TCP)", draft-ietf-tcpm-tcp-security-02 (work in progress), January 2011.

Appendix A. Optional Capability Flags

Certain hosts may want to negotiate a certain optimal Timestamp Clock Rate for various purposes. For example, the balance between PAWS ([RFC1323]) and the timestamp clock resolution should be more towards one or the other. Also, if certain algorithms want to have identical timestamp clock rates both at the sender and receiver, negotiating the clock rate would be preferrable. However, without a full three way handshake, full negotiation of the timestamp clock rate is not possible.

For this purpose, the following extension of this proposal is suggested.

Kind: 8

Length: 10 bytes

```
       +-------+-------+---------------------+---------------------+
       |Kind=8 |  10   |   TS Value (TSval)  |TS Echo Reply (TSecr)|
       +-------+-------+---------------------+---------------------+
           1       1              4          |           4         |
                                            /                      |
       .-----------------------------------'                       |
      /                                                             \
     |                                                               |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |E|R|R|         |         |             |         |             |
     |X|E|N|  MASK   | EXP12lo |  FRAC12lo   | EXP12hi |  FRAC12hi   |
     |O|S|G|         |         |             |         |             |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

timestamp option flags

The following additonal fields are defined:

RNG - Range negotiation
    Indicated that the sender is capable of adjusting the timestamp clock rate within the bounds of the two 12 bit fields (see Appendix A.1). This flag is only valid in the initial SYN segment, and invalid when the ACK bit is set.

EXP12lo and

EXP12hi - binary12 Exponent
    The exponent component of a truncated, 12 bit floating point number indicating the possible timestamp clock ranges. The exponent bias is also 28, and no special numbers (infinity, NaN)

are allowed. The exponent value 31 is treated like any other exponent value. Only the host initiating a TCP session MAY offer a timestamp clock range, while the receiver SHOULD select a timestamp clock within these bounds. If the receiver can not adjust it's timestamp clock to match the range, it MAY use a timestamp clock rate outside these bounds. If the receiver indicated a timestamp clock rate within the indicated bounds, the sender MUST set it's timestamp clock rate to the negotiated rate. If the receiver uses a timestamp clock rate outside the indicated bounds, the sender MUST set the local timestamp clock rate to the value indicated at the closer bound.

FRAC12lo and

FRAC12hi - binary12 Fraction
The fraction component of a 12 bit floating point number. Subnormal numbers are allowed (Exponent value 0). This allows a range between 7.45 ns and 15.99 s with full resolution (lower bound is 0.06 ns using subnormal values).

A.1. Range Negotiation

The following sequence would negotiate the timestamp clock rate for both sender and receiver, where both finally know the clock rate of the respective partner.

SYN, TSopt=<X>, TSecr=EXO|RNG|MASK|12bit-lo=1ms|12bit-hi=100ms

SYN,ACK, TSopt=<Y>, TSecr=<X>^EXO|MASK|16bit=10ms

In this example, both hosts would run their respective timestamp clocks with a resolution of 10 ms.

SYN, TSopt=<X>, TSecr=EXO|RNG|MASK|12bit-lo=1ms|12bit-hi=100ms

SYN,ACK, TSopt=<Y>, TSecr=<X>^EXO|MASK|16bit=1000ms

In this example, the sender would run the timestamp clocks with a resolution of 100 ms (closer to the receivers clock rate of 1 sec), while the receiver will have a timestamp clock rate running at 1 sec.

SYN, TSopt=<X>, TSecr=EXO|RNG|MASK|12bit-lo=1ms|12bit-hi=100ms

SYN,ACK, TSopt=<Y>, TSecr=<X>^EXO|MASK|16bit=100us

In this example, the sender would run the timestamp clocks with a resolution of 10 ms (closer to the receivers clock rate of 0.1 ms), while the receiver will have a timestamp clock rate running at 0.1ms.

Appendix B. Revision history

00 ... initial draft, early submission to meet deadline

01 ... refined draft, focusing only on those options that have an immediate use case. Also excluding flags that can be substituted by other means (MIR - synergistic with SACK options only, RNG moved to appendix, BIA removed while the exponent bias is at a fixed value. Also extended other paragraphs.

Authors' Addresses

Richard Scheffenegger (editor)
NetApp, Inc.
Am Euro Platz 2
Vienna, 1120
Austria

Phone: +43 1 3676811 3146
Email: rs@netapp.com

Mirja Kuehlewind
University of Stuttgart
Pfaffenwaldring 47
Stuttgart 70569
Germany

Email: mirja.kuehlewind@ikr.uni-stuttgart.de