**Survey of Security Hardening Methods for Transmission Control Protocol (TCP) Implementations**
**draft-ietf-tcpm-tcp-security-03.txt**

Abstract

   This document surveys methods to harden Transmission Control Protocol
   (TCP) implementations.  It provides an overview of known attacks and
   refers to the corresponding solutions in the TCP standards.

Status of this Memo

Copyright Notice

Table of Contents

# 1.  Preface

## 1.1.  Introduction

The TCP/IP protocol suite was conceived in an environment that was
quite different from the hostile environment they currently operate
in.  However, the effectiveness of the protocols led to their early
adoption in production environments, to the point that, to some
extent, the current world's economy depends on them.

While many textbooks and articles have created the myth that the
Internet protocols were designed for warfare environments, the top
level goal for the DARPA Internet Program was the sharing of large
service machines on the ARPANET [Clark, 1988].  As a result, many
protocol specifications focus only on the operational aspects of the
protocols they specify, and overlook their security implications.

While the Internet technology evolved since it early inception, the
Internet's building blocks are basically the same core protocols
adopted by the ARPANET more than two decades ago.  During the last
twenty years, many vulnerabilities have been identified in the TCP/IP
stacks of a number of systems.  Some of them were based on flaws in
some protocol implementations, affecting only a reduced number of
systems, while others were based in flaws in the protocols
themselves, affecting virtually every existing implementation
[Bellovin, 1989].  Even in the last couple of years, researchers were
still working on security problems in the core protocols [NISCC,
2004] [NISCC, 2005].

The discovery of vulnerabilities in the TCP/IP protocol suite usually
led to reports being published by a number of CSIRTs (Computer
Security Incident Response Teams) and vendors, which helped to raise
awareness about the threats and the best mitigations known at the
time the reports were published.  Unfortunately, this also led to the
documentation of the discovered protocol vulnerabilities being spread
among a large number of documents, which are sometimes difficult to
identify.

For some reason, much of the effort of the security community on the
Internet protocols did not result in official documents (RFCs) being
issued by the IETF (Internet Engineering Task Force).  This basically
led to a situation in which "known" security problems have not always
been addressed by all vendors.  In addition, in many cases vendors
have implemented quick "fixes" to the identified vulnerabilities
without a careful analysis of their effectiveness and their impact on
interoperability [Silbersack, 2005].

Producing a secure TCP/IP implementation nowadays is a very difficult

task, in part because of the lack of a single document that serves as
a security roadmap for the protocols.  Implementers are faced with
the hard task of identifying relevant documentation and
differentiating between that which provides correct advice, and that
which provides misleading advice based on inaccurate or wrong
assumptions.

This document is the result of a security assessment of the IETF
specifications of the Transmission Control Protocol (TCP), from a
security point of view.  Possible threats are identified and, where
possible, countermeasures are described.  Additionally, many
implementation flaws that have led to security vulnerabilities have
been referenced in the hope that future implementations will not
incur the same problems.

This document is based on the "Security Assessment of the
Transmission Control Protocol (TCP)" released by the UK Centre for
the Protection of National Infrastructure (CPNI), available at: http:
//www.cpni.gov.uk/Products/technicalnotes/
Feb-09-security-assessment-TCP.aspx .

## 1.2.  Scope of this document

While there are a number of protocols that may affect the way TCP
operates, this document focuses only on the specifications of the
Transmission Control Protocol (TCP) itself.

The machanisms described in the following documents were selected for
assessment as part of this work:

o  RFC 793, "Transmission Control Protocol.  DARPA Internet Program.
   Protocol Specification" (91 pages)

o  RFC 1122, "Requirements for Internet Hosts -- Communication
   Layers" (116 pages)

o  RFC 1191, "Path MTU Discovery" (19 pages)

o  RFC 1323, "TCP Extensions for High Performance" (37 pages)

o  RFC 1948, "Defending Against Sequence Number Attacks" (6 pages)

o  RFC 1981, "Path MTU Discovery for IP version 6" (15 pages)

o  RFC 2018, "TCP Selective Acknowledgment Options" (12 pages)

o  RFC 2385, "Protection of BGP Sessions via the TCP MD5 Signature
   Option" (6 pages)

   o  RFC 2581, "TCP Congestion Control" (14 pages)

   o  RFC 2675, "IPv6 Jumbograms" (9 pages)

   o  RFC 2883, "An Extension to the Selective Acknowledgement (SACK)
      Option for TCP" (17 pages)

   o  RFC 2884, "Performance Evaluation of Explicit Congestion
      Notification (ECN) in IP Networks" (18 pages)

   o  RFC 2988, "Computing TCP's Retransmission Timer" (8 pages)

   o  RFC 3168, "The Addition of Explicit Congestion Notification (ECN)
      to IP" (63 pages)

   o  RFC 3465, "TCP Congestion Control with Appropriate Byte Counting
      (ABC)" (10 pages)

   o  RFC 3517, "A Conservative Selective Acknowledgment (SACK)-based
      Loss Recovery Algorithm for TCP" (13 pages)

   o  RFC 3540, "Robust Explicit Congestion Notification (ECN) Signaling
      with Nonces" (13 pages)

   o  RFC 3782, "The NewReno Modification to TCP's Fast Recovery
      Algorithm" (19 pages)

## 1.3.  Organization of this document

   This document is basically organized in two parts.  The first part
   contains a discussion of each of the TCP header fields, identifies
   their security implications, and discusses the possible
   countermeasures.  The second part contains an analysis of the
   security implications of the mechanisms and policies implemented by
   TCP, and of a number of implementation strategies in use by a number
   of popular TCP implementations.

## 2.  The Transmission Control Protocol

   The Transmission Control Protocol (TCP) is a connection-oriented
   transport protocol that provides a reliable byte-stream data transfer
   service.  Very few assumptions are made about the reliability of
   underlying data transfer services below the TCP layer.  Basically,
   TCP assumes it can obtain a simple, potentially unreliable datagram
   service from the lower level protocols.

   The core TCP specification, RFC 793 [RFC0793], dates back to 1981 and

standardizes the basic mechanisms and policies of TCP.  RFC 1122
[RFC1122] provides clarifications and errata for the original
specification.  RFC 2581 [RFC5681] specifies TCP congestion control
and avoidance mechanisms, not present in the original specification.
Other documents specify extensions and improvements for TCP.

The large amount of documents that specify extensions, improvements,
or modifications to existing TCP mechanisms has led the IETF to
publish a roadmap for TCP, RFC 4614 [Duke et al, 2006], that
clarifies the relevance of each of those documents.


## 3.  TCP header fields

RFC 793 [RFC0793] defines the syntax of a TCP segment, along with the
semantics of each of the header fields.

The minimum TCP header size is 20 bytes, and corresponds to a TCP
segment with no options and no data.  However, a TCP module might be
handed an (illegitimate) "TCP segment" of less than 20 bytes.
Therefore, before doing any processing of the TCP header fields, the
following check should be performed by TCP on the segments handed by
the internet layer:

                         Segment.Size >= 20

If a segment does not pass this check, it should be dropped.

The following subsections contain further sanity checks that should
be performed on TCP segments.

### 3.1.  Source Port and Destination Port

The Source Port field contains a 16-bit number that identifies the
TCP end-point that originated this TCP segment.  The TCP Destination
Port contains a 16-bit number that identifies the destination TCP
end-point of this segment.  In most of the discussion we refer to
client-side (or "ephemeral") port-numbers and server-side port
numbers, since that distinction is what usually affects the
interpretation of a port number.

Most active attacks against ongoing TCP connections require the
attacker to guess or know the four-tuple that identifies the
connection.  As a result, randomization of the TCP ephemeral ports
provides a (partial) mitigation against off-path attacks.  [RFC6056]
provides guidance in this area.

Some implementations have been known to crash when a TCP segment in

which the source end-point (IP Source Address, TCP Source Port) is
the same as the destination end-point (IP Destination Address, TCP
Destination Port). [draft-gont-tcpm-tcp-mirrored-endpoints-00.txt]
describes this issue in detail and provides advice in this area.

While some systems restrict use of the port numbers in the range
0-1024 to privileged users, applications should not grant any trust
based on the port numbers used for a TCP connection.

> Not all systems require superuser privileges to bind port numbers
> in that range.  Besides, with desktop computers such "distinction"
> has generally become irrelevant.

Middle-boxes such as packet filters must not assume that clients use
port numbers from only the Dynamic or Registered port ranges.

> It should also be noted that some clients, such as DNS resolvers,
> are known to use port numbers from the "Well Known Ports" range.
> Therefore, middle-boxes such as packet filters MUST NOT assume
> that clients use port number from only the Dynamic or Registered
> port ranges.

### 3.2.  Sequence number

Predictable sequence numbers allow a variety of attacks against TCP,
such as those described in Section 5.2 and Section 11 of this
document.  This vulnerability was first described in [Morris1985],
and its exploitation was widely publicized about 10 years later
[Shimomura1995].

In order to mitigate this vulnerabilities, some implementations set
the TCP ISN to a PRNG.  However, this has been known to cause
interoperability problems.  [RFC6528] provides advice in this area.

Another security consideration that should be made about TCP sequence
numbers is that they might allow an attacker to count the number of
systems behind a Network Address Translator (NAT) [Srisuresh and
Egevang, 2001].  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 the NAT by establishing a number of TCP
connections (using the public address of the NAT) and indentifying
the number of different sequence number "spaces".  [Gont and
Srisuresh, 2008] provides a detailed discussion of the security
implications of NATs and of the possible mitigations for this and
other issues.

### 3.3.  Acknowledgement Number

If the ACK bit is on, the Acknowledgement Number contains the value
of the next sequence number the sender of this segment is expecting
to receive.  According to RFC 793, the Acknowledgement Number is
considered valid as long as it does not acknowledge the receipt of
data that has not yet been sent.

However, as a result of recent concerns on forgery attacks against
TCP (see Section 11 of this document) [RFC5961] has proposed to
enforce a more strict check on the Acknowledgement Number of segments
that have the ACK bit set.  See for more details.

If the ACK bit is off, the Acknowledgement Number field is not valid.
We recommend TCP implementations to set the Acknowledgement Number to
zero when sending a TCP segment that does not have the ACK bit set
(i.e., a SYN segment).  Some TCP implementations have been known to
fail to set the Acknowledgement Number to zero, thus leaking
information.

TCP Acknowledgements are also used to perform heuristics for loss
recovery and congestion control.  Section 9 of this document
describes a number of ways in which these mechanisms can be
exploited.

### 3.4.  Data Offset

[draft-gont-tcpm-tcp-sanity-checks-00.txt] specifies a number of
sanity checks that should be performed on the Data Offset field.

### 3.5.  Control bits

The following subsections provide a discussion of the different
control bits in the TCP header.  TCP segments with unusual
combinations of flags set have been known in the past to cause
malfunction of some implementations, sometimes to the extent of
causing them to crash [RFC1025] [RFC1379].  These packets are still
usually employed for the purpose of TCP/IP stack fingerprinting.
Section 12.1 contains a discussion of TCP/IP stack fingerprinting.

### 3.5.1.  Reserved (four bits)

These four bits are reserved for future use, and must be zero.  As
with virtually every field, the Reserved field could be used as a
covert channel.  While there exist intermediate devices such as
protocol scrubbers that clear these bits, and firewalls that drop/
reject segments with any of these bits set, these devices should
consider the impact of these policies on TCP interoperability.  For

example, as TCP continues to evolve, all or part of the bits in the
Reserved field could be used to implement some new functionality.  If
some middle-box or end-system implementation were to drop a TCP
segment merely because some of these bits are not set to zero,
interoperability problems would arise.

### 3.5.2.  CWR (Congestion Window Reduced)

The CWR flag, defined in RFC 3168 [Ramakrishnan et al, 2001], is used
as part of the Explicit Congestion Notification (ECN) mechanism.  For
connections in any of the synchronized states, this flag indicates,
when set, that the TCP sending this segment has reduced its
congestion window.

An analysis of the security implications of ECN can be found in
Section 9.3 of this document.

### 3.5.3.  ECE (ECN-Echo)

The ECE flag, defined in RFC 3168 [Ramakrishnan et al, 2001], is used
as part of the Explicit Congestion Notification (ECN) mechanism.

An analysis of the security implications of ECN can be found in
Section 9.3 of this document.

### 3.5.4.  URG

When the URG flag is set, the Urgent Pointer field contains the
current value of the urgent pointer.

Receipt of an "urgent" indication generates, in a number of
implementations (such as those in UNIX-like systems), a software
interrupt (signal) that is delivered to the corresponding process.
In UNIX-like systems, receipt of an urgent indication causes a SIGURG
signal to be delivered to the corresponding process.

A number of applications handle TCP urgent indications by installing
a signal handler for the corresponding signal (e.g., SIGURG).  As
discussed in [Zalewski, 2001b], some signal handlers can be
maliciously exploited by an attacker, for example to gain remote
access to a system.  While secure programming of signal handlers is
out of the scope of this document, we nevertheless raise awareness
that TCP urgent indications might be exploited to abuse poorly-
written signal handlers.

Section 3.9 discusses the security implications of the TCP urgent
mechanism.

### 3.5.5.  ACK

When the ACK bit is one, the Acknowledgment Number field contains the
next sequence number expected, cumulatively acknowledging the receipt
of all data up to the sequence number in the Acknowledgement Number,
minus one.  Section 3.4 of this document describes sanity checks that
should be performed on the Acknowledgement Number field.

TCP Acknowledgements are also used to perform heuristics for loss
recovery and congestion control.  Section 9 of this document
describes a number of ways in which these mechanisms can be
exploited.

### 3.5.6.  PSH

[draft-gont-tcpm-tcp-push-semantics-00.txt] describes a number of
security issues that may arise as a result of the PUSH semantics, and
proposes a number of ways to mitigate these issues.

### 3.5.7.  RST

The RST bit is used to request the abortion (abnormal close) of a TCP
connection.  RFC 793 [RFC0793] suggests that an RST segment should be
considered valid if its Sequence Number is valid (i.e., falls within
the receive window).  However, in response to the security concerns
raised by [Watson, 2004] and [NISCC, 2004], [RFC6429] proposed
stricter validity checks.  Please see [RFC6429] for additional
details.

Section 11.1 of this document describes TCP-based connection-reset
attacks, along with a number of countermeasures to mitigate their
impact.

### 3.5.8.  SYN

The SYN bit is used during the connection-establishment phase, to
request the synchronization of sequence numbers.

There are basically four different vulnerabilities that make use of
the SYN bit: SYN-flooding attacks, connection forgery attacks,
connection flooding attacks, and connection-reset attacks.  They are
described in Section 5.1, Section 5.2, Section 5.3, and Section
11.1.2, respectively, along with the possible countermeasures.

### 3.5.9.  FIN

The FIN flag is used to signal the remote end-point the end of the
data transfer in this direction.  Receipt of a valid FIN segment

(i.e., a TCP segment with the FIN flag set) causes the transition in
the connection state, as part of what is usually referred to as the
"connection termination phase".

The connection-termination phase can be exploited to perform a number
of resource-exhaustion attacks.  Section 6 of this document describes
a number of attacks that exploit the connection-termination phase
along with the possible countermeasures.

## 3.6.  Window

The TCP Window field advertises how many bytes of data the remote
peer is allowed to send before a new advertisement is made.
Theoretically, the maximum transfer rate that can be achieved by TCP
is limited to:

Maximum Transfer Rate = Window / RTT

This means that, under ideal network conditions (e.g., no packet
loss), the TCP Window in use should be at least:

$$Window = 2 * Bandwidth * Delay$$

Using a larger Window than that resulting from the previous equation
will not provide any improvements in terms of performance.

In practice, selection of the most convenient Window size may also
depend on a number of other parameters, such as: packet loss rate,
loss recovery mechanisms in use, etc.

An aspect of the TCP Window that is usually overlooked is the
security implications of its size.  Increasing the TCP window
increases the sequence number space that will be considered "valid"
for incoming segments.  Thus, use of unnecessarily large TCP Window
sizes increases TCP's vulnerability to forgery attacks unnecessarily.

In those scenarios in which the network conditions are known and/or
can be easily predicted, it is recommended that the TCP Window is
never set to a value larger than that resulting from the equations
above.  Additionally, the nature of the application running on top of
TCP should be considered when tuning the TCP window.  As an example,
an H.245 signaling application certainly does not have high
requirements on throughput, and thus a window size of around 4 KBytes
will usually fulfill its needs, while keeping TCP's resistance to
off-path forgery attacks at a decent level.  Some rough measurements
seem to indicate that a TCP window of 4Kbytes is common practice for
TCP connections servicing applications such as BGP.

In principle, a possible approach to avoid requiring administrators
to manually set the TCP window would be to implement an automatic
buffer tuning mechanism, such as that described in [Heffner, 2002].
However, as discussed in Section 7.3.2 of this document these
mechanisms can be exploited to perform other types of attacks.

### 3.6.1.  Security implications arising from closed windows

When a TCP end-point is not willing to receive any more data (before
some of the data that have already been received are consumed), it
will advertise a TCP window of zero bytes.  This will effectively
stop the sender from sending any new data to the TCP receiver.
Transmission of new data will resume when the TCP receiver advertises
a nonzero TCP window, usually with a TCP segment that contains no
data ("an ACK").

> This segment is usually referred to as a "window update", as the
> only purpose of this segment is to update the server regarding the
> new window.

To accommodate those scenarios in which the ACK segment that "opens"
the window is lost, TCP implements a "persist timer" that causes the
TCP sender to query the TCP receiver periodically if the last segment
received advertised a window of zero bytes.  This probe simply
consists of sending one byte of new data that will force the TCP
receiver to send an ACK segment back to the TCP sender, containing
the current TCP window.  Similarly to the retransmission timeout
timer, an exponential back-off is used when calculating the
retransmission timer, so that the spacing between probes increases
exponentially.

A fundamental difference between the "persist timer" and the
retransmission timer is that there is no limit on the amount of time
during which a TCP can advertise a zero window.  This means that a
TCP end-point could potentially advertise a zero window forever, thus
keeping kernel memory at the TCP sender tied to the TCP
retransmission buffer.  This could clearly be exploited as a vector
for performing a Denial of Service (DoS) attack against TCP, such as
that described in Section 7.1 of this document.

Section 7.1 of this document describes a Denial of Service attack
that aims at exhausting the kernel memory used for the TCP
retransmission buffer, along with possible countermeasures.

### 3.7.  Checksum

While in principle there should not be security implications arising
from the Checksum field, due to non-RFC-compliant implementations,

the Checksum can be exploited to detect firewalls, evade network
intrusion detection systems (NIDS), and/or perform Denial of Service
attacks.

If a stateful firewall does not check the TCP Checksum in the
segments it processes, an attacker can exploit this situation to
perform a variety of attacks.  For example, he could send a flood of
TCP segments with invalid checksums, which would nevertheless create
state information at the firewall.  When each of these segments is
received at its intended destination, the TCP checksum will be found
to be incorrect, and the corresponding will be silently discarded.
As these segments will not elicit a response (e.g., an RST segment)
from the intended recipients, the corresponding connection state
entries at the firewall will not be removed.  Therefore, an attacker
may end up tying all the state resources of the firewall to TCP
connections that will never complete or be terminated, probably
leading to a Denial of Service to legitimate users, or forcing the
firewall to randomly drop connection state entries.

If a NIDS does not check the Checksum of TCP segments, an attacker
may send TCP segments with an invalid checksum to cause the NIDS to
obtain a TCP data stream different from that obtained by the system
being monitored.  In order to "confuse" the NIDS, the attacker would
send TCP segments with an invalid Checksum and a Sequence Number that
would overlap the sequence number space being used for his malicious
activity.  FTester [Barisani, 2006] is a tool that can be used to
assess NIDS on this issue.

Finally, an attacker performing port-scanning could potentially
exploit intermediate systems that do not check the TCP Checksum to
detect whether a given TCP port is being filtered by an intermediate
firewall, or the port is actually closed by the host being port-
scanned.  If a given TCP port appeared to be closed, the attacker
would then send a SYN segment with an invalid Checksum.  If this
segment elicited a response (either an ICMP error message or a TCP
RST segment) to this packet, then that response should come from a
system that does not check the TCP checksum.  Since normal host
implementations of the TCP protocol do check the TCP checksum, such a
response would most likely come from a firewall or some other middle-
box.

[Ed3f, 2002] describes the exploitation of the TCP checksum for
performing the above activities.  [US-CERT, 2005d] provides an
example of a TCP implementation that failed to check the TCP
checksum.

## 3.8.  Urgent pointer

   Some implementations have been found to be unable to process TCP
   urgent indications correctly.  [Myst, 1997] originally described how
   TCP urgent indications could be exploited to perform a Denial of
   Service (DoS) attack against some TCP/IP implementations, usually
   leading to a system crash.

   [draft-gont-tcpm-tcp-sanity-checks-00.txt] describes a number of
   sanity checks to be enforced on TCP segments regarding urgent
   indications.  [RFC6093] deprecates the use of urgent indications in
   new applications.

## 3.9.  Options

   [IANA, 2007] contains the official list of the assigned option
   numbers.  TCP Options have been specified in the past both within the
   IETF and by other groups.  [Hnes, 2007] contains an un-official
   updated version of the IANA list of assigned option numbers.  The
   following table contains a summary of the assigned TCP option
   numbers, which is based on [Hnes, 2007].

| Kind | Meaning | Summary |
|------|---------|---------|
| 0 | End of Option List | Discussed in Section 4.1 |
| 1 | No-Operation | Discussed in Section 4.2 |
| 2 | Maximum Segment Size | Discussed in Section 4.3 |
| 3 | WSOPT - Window Scale | Discussed in Section 4.6 |
| 4 | SACK Permitted | Discussed in Section 4.4.1 |
| 5 | SACK | Discussed in Section 4.4.2 |
| 6 | Echo (obsoleted by option 8) | Obsolete.  Specified in RFC 1072 [Jacobson and Braden, 1988] |
| 7 | Echo Reply (obsoleted by option 8) | Obsolete.  Specified in RFC 1072 [Jacobson and Braden, 1988] |
| 8 | TSOPT - Time Stamp Option | Discussed in Section 4.7 |
| 9 | Partial Order Connection Permitted | Historic.  Specified in RFC 1693 [Connolly et al, 1994] |
| 10 | Partial Order Service Profile | Historic.  Specified in RFC 1693 [Connolly et al, 1994] |
| 11 | CC | Historic.  Specified in RFC 1644 [Braden, 1994] |
| 12 | CC.NEW | Historic.  Specified in RFC 1644 [Braden, 1994] |
| 13 | CC.ECHO | Historic.  Specified in RFC 1644 [Braden, 1994] |
| 14 | TCP Alternate Checksum Request | Historic.  Specified in RFC 1146 [Zweig and Partridge, 1990] |
| 15 | TCP Alternate Checksum Data | Historic.  Specified in RFC 1145 [Zweig and Partridge, 1990] |
| 16 | Skeeter | Historic |

| 17 | Bubba | Historic |
|---|---|---|
| 18 | Trailer Checksum Option | Historic |
| 19 | MD5 Signature Option | Discussed in Section 4.5 |
| 20 | SCPS Capabilities | Specified in [CCSDS, 2006] |
| 21 | Selective Negative Acknowledgements | Specified in [CCSDS, 2006] |
| 22 | Record Boundaries | Specified in [CCSDS, 2006] |
| 23 | Corruption experienced | Specified in [CCSDS, 2006] |
| 24 | SNAP | Historic |
| 25 | Unassigned (released 2000-12-18) | Unassigned |
| 26 | TCP Compression Filter | Historic |
| 27 | Quick-Start Response | Specified in RFC 4782 [Floyd et al, 2007] |
| 28-252 | Unassigned | Unassigned |
| 253 | RFC3692-style Experiment 1 | Described by RFC 4727 [Fenner, 2006] |
| 254 | RFC3692-style Experiment 2 | Described by RFC 4727 [Fenner, 2006] |

Table 1: TCP Options

There are two cases for the format of a TCP option:

o  Case 1: A single byte of option-kind.

o  Case 2: An option-kind byte, followed by an option-length byte,
   and the actual option-data bytes.

In options of the Case 2 above, the option-length byte counts the

option-kind byte and the option-length byte, as well as the actual
option-data bytes.

All options except "End of Option List" (Kind = 0) and "No Operation"
(Kind = 1), are of "Case 2".

[draft-gont-tcpm-tcp-sanity-checks-00.txt] describes a number of
sanity checks that should be performed on TCP options.

Section 4 discusses the security implications of common TCP options.

## 3.10.  Padding

The TCP header padding is used to ensure that the TCP header ends and
data begins on a 32-bit boundary.  The padding is composed of zeros.

## 3.11.  Data

The data field contains the upper-layer packet being transmitted by
means of TCP.  This payload is processed by the application process
making use of the transport services of TCP.  Therefore, the security
implications of this field are out of the scope of this document.


## 4.  Common TCP Options

## 4.1.  End of Option List (Kind = 0)

This option indicates the "End of Options".  As noted in
[draft-gont-tcpm-tcp-sanity-checks-00.txt], some implementations pad
the end of options with "No Operation" options rather than including
an "End of Options List" option.

## 4.2.  No Operation (Kind = 1)

The no-operation option is basically used to allow the sending system
to align subsequent options in, for example, 32-bit boundaries.

This option does not have any known security implications.

## 4.3.  Maximum Segment Size (Kind = 2)

The Maximum Segment Size (MSS) option is used to indicate to the
remote TCP endpoint the maximum segment size this TCP is willing to
receive.

The MSS option has been employed for performing DoS attacks, by
advertising very small MSS values thus greatly increasing the packet-

rate used by the victim system.
[draft-gont-tcpm-tcp-sanity-checks-00.txt] describes this issue, and
proposes sanity checks to mitigate it.

## 4.4. Selective Acknowledgement Option

The Selective Acknowledgement option provides an extension to allow
the acknowledgement of individual segments, to enhance TCP's loss
recovery.

Two options are involved in the SACK mechanism.  The "Sack-permitted
option" is sent during the connections-establishment phase, to
advertise that SACK is supported.  If both TCP peers agree to use
selective acknowledgements, the actual selective acknowledgements are
sent, if needed, by means of "SACK options".

### 4.4.1. SACK-permitted Option (Kind = 4)

[draft-gont-tcpm-tcp-sanity-checks-00.txt] to be performed on this
option.

### 4.4.2. SACK Option (Kind = 5)

The TCP receiving a SACK option is expected to keep track of the
selectively-acknowledged blocks.  Even when space in the TCP header
is limited (and thus each TCP segment can selectively-acknowledge at
most four blocks of data), an attacker could try to perform a buffer
overflow or a resource-exhaustion attack by sending a large number of
SACK options.

For example, an attacker could send a large number of SACK options,
each of them acknowledging one byte of data.  Additionally, for the
purpose of wasting resources on the attacked system, each of these
blocks would be separated from each other by one byte, to prevent the
attacked system from coalescing two (or more) contiguous SACK blocks
into a single SACK block.  If the attacked system kept track of each
SACKed block by storing both the Left Edge and the Right Edge of the
block, then for each window of data, the attacker could waste up to 4
* Window bytes of memory at the attacked TCP.

   The value "4 * Window" results from the expression "(Window / 2) *
   8", in which the value "2" accounts for the 1-byte block
   selectively-acknowledged by each SACK block and 1 byte that would
   be used to separate each SACK blocks from each other, and the
   value "8" accounts for the 8 bytes needed to store the Left Edge
   and the Right Edge of each SACKed block.

[draft-gont-tcpm-tcp-sanity-checks-00.txt] describes sanity checks to

be performed on this option such that this and other possible issues
are mitigated.

## 4.5. MD5 Option (Kind=19)

The TCP MD5 option provides a mechanism for authenticating TCP
segments with a 18-byte digest produced by the MD5 algorithm.  The
option consists of an option-kind byte (which must be 19), an option-
length byte (which must be 18), and a 16-byte MD5 digest.

A basic weakness on the TCP MD5 option is that the MD5 algorithm
itself has been known (for a long time) to be vulnerable to collision
search attacks.

[Bellovin, 2006] argues that it has two other weaknesses, namely that
it does not provide a key identifier, and that it has no provision
for automated key management.  However, it is generally accepted that
while a Key-ID field can be a good approach for providing smooth key
rollover, it is not actually a requirement.  For instance, most
systems implementing the TCP MD5 option include a "keychain"
mechanism that fully supports smooth key rollover.  Additionally,
with some further work, ISAKMP/IKE could be used to configure the MD5
keys.

It is interesting to note that while the TCP MD5 option, as specified
by RFC 2385 [Heffernan, 1998], addresses the TCP-based forgery
attacks against TCP discussed in Section 11, it does not address the
ICMP-based connection-reset attacks discussed in Section 15.  As a
result, while a TCP connection may be protected from TCP-based
forgery attacks by means of the MD5 option, an attacker might still
be able to successfully perform the ICMP-based counter-part.

The TCP MD5 option has been obsoleted by the TCP-AO.

## 4.6. Window scale option (Kind = 3)

The window scale option provides a mechanism to expand the definition
of the TCP window to 32 bits, such that the performance of TCP can be
improved in some network scenarios.  The Window scale option consists
of an option-kind byte (which must be 3), followed by an option-
length byte (which must be 3), and a shift count (shift.cnt) byte
(the actual option-data).

While there are not known security implications arising from the
window scale mechanism itself, the size of the TCP window has a
number of security implications.  In general, larger window sizes
increase the chances of an attacker from successfully performing
forgery attacks against TCP, such as those described in Section 11 of

this document.  Additionally, large windows can exacerbate the impact
of resource exhaustion attacks such as those described in Section 7
of this document.

Section 3.7 provides a general discussion of the security
implications of the TCP window size.  Section 7.3.2 discusses the
security implications of Automatic receive-buffer tuning mechanisms.

### 4.7.  Timestamps option (Kind = 8)

The Timestamps option, specified in RFC 1323 [Jacobson et al, 1992],
is used to perform two functions: Round-Trip Time Measurement (RTTM),
and Protection Against Wrapped Sequence Numbers (PAWS).

### 4.7.1.  Generation of timestamps

For the purpose of PAWS, the timestamps sent on a connection are
required to be monotonically increasing.  While there is no
requirement that timestamps are monotonically increasing across TCP
connections, the generation of timestamps such that they are
monotonically increasing across connections between the same two
endpoints allows the use of timestamps for improving the handling of
SYN segments that are received while the corresponding four-tuple is
in the TIME-WAIT state.  This is discussed in Section 11.1.2 of this
document.

Some implementations are known to initialize their global timestamp
clock to zero when the system is bootstrapped.  This is undesirable,
as the timestamp clock would disclose the system uptime.
[I-D.gont-timestamps-generation] discusses the generation of TCP
timestamps in detail.

### 4.7.2.  Vulnerabilities

Blind In-Window Attacks

Segments that contain a timestamp option smaller than the last
timestamp option recorded by TCP are silently dropped.  This allows
for a subtle attack against TCP that would allow an attacker to cause
one direction of data transfer of the attacked connection to freeze
[US-CERT, 2005c].  An attacker could forge a TCP segment that
contains a timestamp that is much larger than the last timestamp
recorded for that direction of the data transfer of the connection.
The offending segment would cause the recorded timestamp (TS.Recent)
to be updated and, as a result, subsequent segments sent by the
impersonated TCP peer would be simply dropped by the receiving TCP.
This vulnerability has been documented in [US-CERT, 2005d].  However,
it is worth noting that exploitation of this vulnerability requires

an attacker to guess (or know) the four-tuple {IP Source Address, IP
Destination Address, TCP Source Port, TCP Destination Port}, as well
a valid Sequence Number and a valid Acknowledgement Number.  If an
attacker has such detailed knowledge about a TCP connection, unless
TCP segments are protected by proper authentication mechanisms (such
as IPsec [Kent and Seo, 2005]), he can perform a variety of attacks
against the TCP connection, even more devastating than the one just
described.

Information leaking

Some implementations are known to maintain a global timestamp clock,
which is used for all connections.  This is undesirable, as an
attacker that can establish a connection with a host would learn the
timestamp used for all the other connections maintained by that host,
which could be useful for performing any attacks that require the
attacker to forge TCP segments.  A timestamps generator such as the
one recommended in Section 4.7.1 of this document would prevent this
information leakage, as it separates the "timestamps space" among the
different TCP connections.

Some implementations are known to initialize their global timestamp
clock to zero when the system is bootstrapped.  This is undesirable,
as the timestamp clock would disclose the system uptime.  A
timestamps generator such as the one recommended in Section 4.7.1 of
this document would prevent this information leakage, as the function
F() introduces an "offset" that does not disclose the system uptime.

As discussed in Section 3.2 of RFC 1323 [Jacobson et al, 1992], the
Timestamp Echo Reply field (TSecr) is only valid if the ACK bit of
the TCP header is set, and its value must be zero when it is not
valid.  However, some TCP implementations have been found to fail to
set the Timestamp Echo Reply field (TSecr) to zero in TCP segments
that do not have the ACK bit set, thus potentially leaking
information.  We stress that TCP implementations should comply with
RFC 1323 by setting the Timestamp Echo Reply field (TSecr) to zero in
those TCP segments that do not have the ACK bit set, thus eliminating
this potential information leakage.

Finally, it should be noted that the Timestamps option can be
exploited to count the number of systems behind NATs (Network Address
Translators) [Srisuresh and Egevang, 2001].  An attacker could 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 timestamp sequences.  This information
leakage could be eliminated by rewriting the contents of the
Timestamps option at the NAT.  [Gont and Srisuresh, 2008] provides a
detailed discussion of the security implications of NATs, and

proposes mitigations for this and other issues.

## 5.  Connection-establishment mechanism

The following subsections describe a number of attacks that can be
performed against TCP by exploiting its connection-establishment
mechanism.

### 5.1.  SYN flood

TCP uses a mechanism known as the "three-way handshake" for the
establishment of a connection between two TCP peers.  RFC 793
[RFC0793] states that when a TCP that is in the LISTEN state receives
a SYN segment (i.e., a TCP segment with the SYN flag set), it must
transition to the SYN-RECEIVED state, record the control information
(e.g., the ISN) contained in the SYN segment in a Transmission
Control Block (TCB), and respond with a SYN/ACK segment.

A Transmission Control Block is the data structure used to store
(usually within the kernel) all the information relevant to a TCP
connection.  The concept of "TCB" is introduced in the core TCP
specification RFC 793 [RFC0793].

In practice, virtually all existing implementations do not modify the
state of the TCP that was in the LISTEN state, but rather create a
new TCP (i.e., a new "protocol machine"), and perform all the state
transitions on this newly-created TCP.  This allows the application
running on top of TCP to service to more than one client at the same
time.  As a result, each connection request results in the allocation
of system memory to store the TCB associated with the newly created
TCB.

If TCP was implemented strictly as described in RFC 793, the
application running on top of TCP would have to finish servicing the
current client before being able to service the next one in line, or
should instead be able to perform some kind of connection hand-off.

An attacker could exploit TCP's connection-establishment mechanism to
perform a Denial of Service (DoS) attack, by sending a large number
of connection requests to the target system, with the intent of
exhausting the system memory destined for storing TCBs (or related
kernel data structures), thus preventing the attacked system from
establishing new connections with legitimate users.  This attack is
widely known as "SYN flood", and has received a lot of attention
during the late 90's [CERT, 1996].

Given that the attacker does not need to complete the three-way

handshake for the attacked system to tie system resources to the
newly created TCBs, he will typically forge the source IP address of
the malicious SYN segments he sends, thus concealing his own IP
address.

If the forged IP addresses corresponded to some reachable system, the
impersonated system would receive the SYN/ACK segment sent by the
attacked host (in response to the forged SYN segment), which would
elicit an RST segment.  This RST segment would be delivered to the
attacked system, causing the corresponding connection to be aborted,
and the corresponding TCB to be removed.

As the impersonated host would not have any state information for the
TCP connection being referred to by the SYN/ACK segment, it would
respond with a RST segment, as specified by the TCP segment
processing rules of RFC 793 [RFC0793].

However, if the forged IP source addresses were unreachable, the
attacked TCP would continue retransmitting the SYN/ACK segment
corresponding to each connection request, until timing out and
aborting the connection.  For this reason, a number of widely
available attack tools first check whether each of the (forged) IP
addresses are reachable by sending an ICMP echo request to them.  The
receipt of an ICMP echo response is considered an indication of the
IP address being reachable (and thus results in the corresponding IP
address not being used for performing the attack), while the receipt
of an ICMP unreachable error message is considered an indication of
the IP address being unreachable (and thus results in the
corresponding IP address being used for performing the attack).

[Gont, 2008b] describes how the so-called ICMP soft errors could be
used by TCP to abort connections in any of the non-synchronized
states.  While implementation of the mechanism described in that
document would certainly not eliminate the vulnerability of TCP to
SYN flood attacks (as the attacker could use addresses that are
simply "black-holed"), it provides an example of how signaling
information such as that provided by means of ICMP error messages can
provide valuable information that a transport protocol could use to
perform heuristics.

In order to mitigate the impact of this attack, the amount of
information stored for non-established connections should be reduced
(ideally, non-synchronized connections should not require any state
information to be maintained at the TCP performing the passive OPEN).
There are basically two mitigation techniques for this vulnerability:
a syn-cache and syn-cookies.

[Borman, 1997] and RFC 4987 [Eddy, 2007] contain a general discussion

of SYN-flooding attacks and common mitigation approaches.

The syn-cache [Lemon, 2002] approach aims at reducing the amount of
state information that is maintained for connections in the SYN-
RECEIVED state, and allocates a full TCB only after the connection
has transited to the ESTABLISHED state.

The syn-cookie [Bernstein, 1996] approach aims at completely
eliminating the need to maintain state information at the TCP
performing the passive OPEN, by encoding the most elementary
information required to complete the three-way handshake in the
Sequence Number of the SYN/ACK segment that is sent in response to
the received SYN segment.  Thus, TCP is relieved from keeping state
for connections in the SYN-RECEIVED state.

The syn-cookie approach has a number of drawbacks:

o  Firstly, given the limited space in the Sequence Number field, it
   is not possible to encode all the information included in the
   initial segment, such as, for example, support of Selective
   Acknowledgements (SACK).

o  Secondly, in the event that the Acknowledgement segment sent in
   response to the SYN/ACK sent by the TCP that performed the passive
   OPEN (i.e., the TCP server) were lost, the connection would end up
   in the ESTABLISHED state on the client-side, but in the CLOSED
   state on the server side.  This scenario is normally handled in
   TCP by having the TCP server retransmit its SYN/ACK.  However, if
   syn-cookies are enabled, there would be no connection state
   information on the server side, and thus the SYN/ACK would never
   be retransmitted.  This could lead to a scenario in which the
   connection could remain in the ESTABLISHED state on the client
   side, but in the CLOSED state at the server side, indefinitely.
   If the application protocol was such that it required the client
   to wait for some data from the server (e.g., a greeting message)
   before sending any data to the server, a deadlock would take
   place, with the client application waiting for such server data,
   and the server waiting for the TCP three-way handshake to
   complete.

o  Thirdly, unless the function used to encode information in the
   SYN/ACK packet is cryptographically strong, an attacker could
   forge TCP connections in the ESTABLISHED state by forging ACK
   segments that would be considered as "legitimate" by the receiving
   TCP.

o  Fourthly, in those scenarios in which establishment of new
   connections is blocked by simply dropping segments with the SYN

bit set, use of SYN cookies could allow an attacker to bypass the firewall rules, as a connection could be established by forging an ACK segment with the correct values, without the need of setting the SYN bit.

As a result, syn-cookies are usually not employed as a first line of defense against SYN-flood attacks, but are only as the last resort to cope with them.  For example, some TCP implementations enable syn-cookies only after a certain number of TCBs has been allocated for connections in the SYN-RECEIVED state.  We recommend this implementation technique, with a syn-cache enabled by default, and use of syn-cookies triggered, for example, when the limit of TCBs for non-synchronized connections with a given port number has been reached.

It is interesting to note that a SYN-flood attack should only affect the establishment of new connections.  A number of books and online documents seem to assume that TCP will not be able to respond to any TCP segment that is meant for a TCP port that is being SYN-flooded (e.g., respond with an RST segment upon receipt of a TCP segment that refers to a non-existent TCP connection).  While SYN-flooding attacks have been successfully exploited in the past for achieving such a goal [Shimomura, 1995], as clarified by RFC 1948 [Bellovin, 1996] the effectiveness of SYN flood attacks to silence a TCP implementation arose as a result of a bug in the 4.4BSD TCP implementation [Wright and Stevens, 1994], rather than from a theoretical property of SYN-flood attacks themselves.  Therefore, those TCP implementations that do not suffer from such a bug should not be silenced as a result of a SYN-flood attack.

[Zquete, 2002] describes a mechanism that could theoretically improve the functionality of SYN cookies.  It exploits the TCP "simultaneous open" mechanism, as illustrated in Figure 5.

        See Figure 5, in page 46 of the UK CPNI document.

        Use of TCP simultaneous open for handling SYN floods

In line 1, TCP A initiates the connection-establishment phase by sending a SYN segment to TCP B. In line 2, TCP B creates a SYN cookie as described by [Bernstein, 1996], but does not set the ACK bit of the segment it sends (thus really sending a SYN segment, rather than a SYN/ACK).  This "fools" TCP A into thinking that both SYN segments "have crossed each other in the network" as if a "simultaneous open" scenario had taken place.  As a result, in line 3 TCP A sends a SYN/ACK segment containing the same options that were contained in the original SYN segment.  In line 4, upon receipt of this segment, TCP processes the cookie encoded in the ACK field as if it had been the

result of a traditional SYN cookie scenario, and moves the connection
into the ESTABLISHED state.  In line 5, TCP B sends a SYN/ACK
segment, which causes the connection at TCP A to move into the
ESTABLISHED state.  In line 6, TCP A sends a data segment on the
connection.

While this mechanism would work in theory, unfortunately there are a
number of factors that prevent it from being usable in real network
environments:

o  Some systems are not able to perform the "simultaneous open"
   operation specified in RFC 793, and thus the connection
   establishment will fail.

o  Some firewalls might prevent the establishment of TCP connections
   that rely on the "simultaneous open" mechanism (e.g., a given
   firewall might be allowing incoming SYN/ACK segments, but not
   outgoing SYN/ACK segments).

Therefore, we do not recommend implementation of this mechanism for
mitigating SYN-flood attacks.

## 5.2.  Connection forgery

The process of causing a TCP connection to be illegitimately
established between two arbitrary remote peers is usually referred to
as "connection spoofing" or "connection forgery".  This can have a
great negative impact when systems establish some sort of trust
relationships based on the IP addresses used to establish a TCP
connection [daemon9 et al, 1996].

It should be stressed that hosts should not establish trust
relationships based on the IP addresses [CPNI, 2008] or on the TCP
ports in use for the TCP connection (see Section 3.1 and Section 3.2
of this document).

One of the underlying weaknesses that allow this vulnerability to be
more easily exploited is the use of an inadequate Initial Sequence
Number (ISN) generator, as explained back in the 80's in [Morris,
1985].  As discussed in Section 3.3.1 of this document, any TCP
implementation that makes use of an inadequate ISN generator will be
more vulnerable to this type of attack.  A discussion of approaches
for a more careful generation of Initial Sequence Numbers (ISNs) can
be found in Section 3.3.1 of this document.

Another attack vector for performing connection-forgery attacks is
the use of IP source routing.  By forging the Source Address of the
IP packets that encapsulate the TCP segments of a connection, and

carefully crafting an IP source route option (i.e., either LSSR or
SSRR) that includes a system whose traffic he can monitor, an
attacker could cause the packets sent by the attacked system (e.g.,
the SYN/ACK segment sent in response to the attacker's SYN segment)
to be illegitimately directed to him [CPNI, 2008].  Thus, the
attacker would not even need to guess valid sequence numbers for
forging a TCP connection, as he would simply have direct access to
all this information.  As discussed in [CPNI, 2008], it is strongly
recommended that systems disable IP Source Routing by default, or at
the very least, they disable source routing for IP packets that
encapsulate TCP segments.

The IPv6 Routing Header Type 0, which provides a similar
functionality to that provided by IPv4 source routing, has been
officially deprecated by RFC 5095 [Abley et al, 2007].

## 5.3.  Connection-flooding attack

NOTE: THIS SECTION IS BEING EDITED.  RFC2119-LANGUAGE IS BEING
REMOVED.

### 5.3.1.  Vulnerability

The creation and maintenance of a TCP connection requires system
memory to maintain shared state between the local and the remote TCP.
As system memory is a finite resource, there is a limit on the number
of TCP connections that a system can maintain at any time.  When the
TCP API is employed to create a TCP connection with a remote peer, it
allocates system memory for maintaining shared state with the remote
TCP peer, and thus the resulting connection would tie a similar
amount of resources at the remote host as at the local host.
However, if special packet-crafting tools are employed to forge TCP
segments to establish TCP connections with a remote peer, the local
kernel implementation of TCP can be bypassed, and the allocation of
resources on the attacker's system for maintaining shared state can
be avoided.  Thus, a malicious user could create a large number of
TCP connections, and subsequently abandon them, thus tying system
resources only at the remote peer.  This allows an attacker to create
a large number of TCP connections at the attacked system with the
intent of exhausting its kernel memory, without exhausting the
attacker's own resources.  [CERT, 2000] discusses this vulnerability,
which is usually referred to as the "Naptha attack".

This attack is similar in nature to the "Netkill" attack discussed in
Section 7.1.1.  However, while Netkill ties both TCBs and TCP send
buffers to the abandoned connections, Naptha only ties TCBs (and
related kernel structures), as it doesn't issue any application
requests.

The symptom of this attack is an extremely large number of TCP
connections in the ESTABLISHED state, which would tend to exhaust
system resources and deny service to new clients (or possibly cause
the system to crash).

It should be noted that it is possible for an attacker to perform the
same type of attack causing the abandoned connections to remain in
states other than ESTABLISHED.  This might be interesting for an
attacker, as it is usually the case that connections in states other
than ESTABLISHED usually have no controlling user-space process (that
is, the former controlling process for the connection has already
closed the corresponding file descriptor).

A particularly interesting case of a connection-flooding attack that
aims at abandoning connections in a state other than ESTABLISHED is
discussed in Section 6.1 of this document.

## 5.3.2.  Countermeasures

As with many other resource exhaustion attacks, the problem in
generating countermeasures for this attack is that it may be
difficult to differentiate between an actual attack and a legitimate
high-load scenario.  However, there are a number of countermeasures
which, when tuned for each particular network environment, could
allow a system to resist this attack and continue servicing
legitimate clients.

Hosts SHOULD enforce limits on the number of TCP connections with no
user-space controlling process.

DISCUSSION:

   Connections in states other than ESTABLISHED usually have no user-
   space controlling process.  This prevents the application making
   use of those connections from enforcing limits on the maximum
   number of ongoing connections (either on a global basis or a
   per-IP address basis).  When resource exhaustion is imminent or
   some threshold of ongoing connections is reached, the operating
   system should consider freeing system resources by aborting
   connections that have no user-space controlling process.  A number
   of such connections could be aborted on a random basis, or based
   on some heuristics performed by the operating system (e.g., first
   abort connections with peers that have the largest number of
   ongoing connections with no user-space controlling process).

Hosts SHOULD enforce per-process and per-user limits on maximum
kernel memory that can be used at any time.

Hosts SHOULD enforce per-process and per-user limits on the number of
existent TCP connections at any time.

DISCUSSION:

   While the Naphta attack is usually targeted at a service such as
   HTTP, its impact is usually system-wide.  This is particularly
   undesirable, as an attack against a single service might affect
   the system as a whole (for example, possibly precluding remote
   system administration).

   In order to avoid an attack to a single service from affecting
   other services, we advise TCP implementations to enforce per-
   process and per-user limits on maximum kernel memory that can be
   used at any time.  Additionally, we recommend implementations to
   enforce per-process and per-user limits on the number of existent
   TCP connections at any time.

Applications SHOULD enforce limits on the number of simultaneous
connections that can be established from a single IP address or
network prefix at any given time.

DISCUSSION:

   An application could limit the number of simultaneous connections
   that can be established from a single IP address or network prefix
   at any given time.  Once that limit has been reached, some other
   connection from the same IP address or network prefix would be
   aborted, thus allowing the application to service this new
   incoming connection.

   There are a number of factors that should be taken into account
   when defining the specific limit to enforce.  For example, in the
   case of protocols that have an authentication phase (e.g., SSH,
   POP3, etc.), this limit could be applied to sessions that have not
   yet been authenticated.  Additionally, depending on the nature and
   use of the application, it might or might not be normal for a
   single system to have multiple connections to the same server at
   the same time.

   For many network services, the limit of maximum simultaneous
   connections could be kept very low.  For example, an SMTP server
   could limit the number of simultaneous connections from a single
   IP address to 10 or 20 connections.

   While this limit could work in many network scenarios, we
   recommend network operators to measure the maximum number of
   concurrent connections from a single IP address during normal

operation, and set the limit accordingly.

In the case of web servers, this limit will usually need to be set
much higher, as it is common practice for web clients to establish
multiple simultaneous connections with a single web server to
speed up the process of loading a web page (e.g., multiple graphic
files can be downloaded simultaneously using separate TCP
connections).

NATs (Network Address Translators) [Srisuresh and Egevang, 2001]
are widely deployed in the Internet, and may exacerbate this
situation, as a large number of clients behind a NAT might each
establish multiple TCP connections with a given web server, which
would all appear to be originate from the same IP address (that of
the NAT box).

Firewalls MAY enforce limits on the number of simultaneous
connections that can be established from a single IP address or
network prefix at any given time.

DISCUSSION:

Some firewalls can be configured to limit the number of
simultaneous connections that any system can maintain with a
specific system and/or service at any given time.  Limiting the
number of simultaneous connections that each system can establish
with a specific system and service would effectively limit the
possibility of an attacker that controls a single IP address to
exhaust system resources at the attacker system/service.

## 5.4.  Firewall-bypassing techniques

[draft-gont-tcpm-tcp-sanity-checks-00.txt] discusses how packets with
both the SYN and RST bits set have been employed in the wild to
bypass firewall rules, and provides advices in this area.


## 6.  Connection-termination mechanism

## 6.1.  FIN-WAIT-2 flooding attack

## 6.1.1.  Vulnerability

TCP implements a connection-termination mechanism that is employed
for the graceful termination of a TCP connection.  This mechanism
usually consists of the exchange of four-segments.  Figure 6
illustrates the usual segment exchange for this mechanism.

Figure 6: TCP connection-termination mechanism

           See Figure 6, in page 50 of the UK CPNI document.

                  TCP connection-termination mechanism

A potential problem may arise as a result of the FIN-WAIT-2 state:
there is no limit on the amount of time that a TCP can remain in the
FIN-WAIT-2 state.  Furthermore, no segment exchange is required to
maintain the connection in that state.

As a result, an attacker could establish a large number of
connections with the target system, and cause it close each of them.
For each connection, once the target system has sent its FIN segment,
the attacker would acknowledge the receipt of this segment, but would
send no further segments on that connection.  As a result, an
attacker could cause the corresponding system resources (e.g., the
system memory used for storing the TCB) without the need to send any
further packets.

While the CLOSE command described in RFC 793 [RFC0793] simply signals
the remote TCP end-point that this TCP has finished sending data
(i.e., it closes only one direction of the data transfer), the
close() system-call available in most operating systems has different
semantics: it marks the corresponding file descriptor as closed (and
thus it is no longer usable), and assigns the operating system the
responsibility to deliver any queued data to the remote TCP peer and
to terminate the TCP connection.  This makes the FIN-WAIT-2 state
particularly attractive for performing memory exhaustion attacks, as
even if the application running on top of TCP were imposing limits on
the maximum number of ongoing connections, and/or time limits on the
function calls performed on TCP connections, that application would
be unable to enforce these limits on the FIN-WAIT-2 state.

## 6.1.2.  Countermeasures

A number of countermeasures can be implemented to mitigate FIN-WAIT-2
flooding attacks.  Some of these countermeasures require changes in
the TCP implementations, while others require changes in the
applications running on top of TCP.

TCP SHOULD enforce limits on the duration of the FIN-WAIT-2 state.

DISCUSSION:

   In order to avoid the risk of having connections stuck in the FIN-
   WAIT-2 state indefinitely, a number of systems incorporate a
   timeout for the FIN-WAIT-2 state.  For example, the Linux kernel

version 2.4 enforces a timeout of 60 seconds [Linux, 2008].  If
the connection-termination mechanism does not complete before that
timeout value, it is aborted.

Enabling applications to enforce limits on ongoing connections

As discussed in Section 6.1.1, the fact that the close() system call
marks the corresponding file descriptor as closed prevents the
application running on top of TCP from enforcing limits on the
corresponding connection.

While it is common practice for applications to terminate their
connections by means of the close() system call, it is possible for
an application to initiate the connection-termination phase without
closing the corresponding file descriptor (hence keeping control of
the connection).

In order to achieve this, an application performing an active close
(i.e., initiating the connection-termination phase) should replace
the system-call close(sockfd) with the following code sequence:

o  A call to shutdown(sockfd, SHUT_WR), to close the sending
   direction of this connection

o  Successive calls to read(), until it returns 0, thus indicating
   that the remote TCP peer has finished sending data.

o  A call to setsockopt(sockfd, SOL_SOCKET, SO_LINGER, &l,
   sizeof(l)), where l is of type struct linger (with its members
   l.l_onoff=1 and l.l_linger=90).

o  A call to close(sockfd), to close the corresponding file
   descriptor.

The call to shutdown() (instead of close()) allows the application to
retain control of the underlying TCP connection while the connection
transitions through the FIN-WAIT-1 and FIN-WAIT-2 states.  However,
the application will not retain control of the connection while it
transitions through the CLOSING and TIME-WAIT states.

It should be noted that, strictly speaking, close(sockfd) decrements
the reference count for the descriptor sockfd, and initiates the
connection termination phase only when the reference count reaches 0.
On the other hand, shutdown(sockfd, SHUT_WR) initiates the
connection-termination phase, regardless of the reference count for
the sockfd descriptor.  This should be taken into account when
performing the code replacement described above.  For example, it
would be a bug for two processes (e.g., parent and child) that share

a descriptor to both call shutdown(sockfd, SHUT_WR).

An application performing a passive close should replace the call to close(sockfd) with the following code sequence:

o  A call to setsockopt(sockfd, SOL_SOCKET, SO_LINGER, &l, sizeof(l)), where l is of type struct linger (with its members l.l_onoff=1 and l.l_linger=90).

o  A call to close(sockfd), to close the corresponding file descriptor.

It is assumed that if the application is performing a passive close, the application already detected that the remote TCP peer finished sending data by means as a result of a call to read() returning 0.

In this scenario, the application will not retain control of the underlying connection when it transitions through the LAST_ACK state.

Enforcing limits on the number of connections with no user-space controlling process

The considerations and recommendations in Section 5.3.2 for enforcing limits on the number of connections with no user-space controlling process are applicable to mitigate this vulnerability.

Limiting the number of simultaneous connections at the application

The considerations and recommendations in Section 5.3.2 for limiting the number of simultaneous connections at the application are to mitigate this vulnerability.  We note, however, that unless applications are implemented to retain control of the underlying TCP connection while the connection transitions through the FIN-WAIT-1 and FIN-WAIT-2 states, enforcing such limits may prove to be a difficult task.

Limiting the number of simultaneous connections at firewalls

The considerations and recommendations in Section 5.3.2 for enforcing limiting the number of simultaneous connections at firewalls are applicable to mitigate this vulnerability.


## 7.  Buffer management

**7.1**.  **TCP retransmission buffer**

**7.1.1**.  **Vulnerability**

   [Shalunov, 2000] describes a resource exhaustion attack (Netkill)
   that can be performed against TCP.  The attack aims at exhausting
   system memory by creating a large number of TCP connections which are
   then abandoned.  The attack is usually performed as follows:

   o  The attacker creates a TCP connection to a service in which a
      small client request can result in a large server response (e.g.,
      HTTP).  Rather than relying on his kernel implementation of TCP,
      the attacker creates his TCP connections by means of a specialized
      packet-crafting tool.  This allows the attacker to create the TCP
      connections and later abandon them, exhausting the resources at
      the attacked system, while not tying his own system resources to
      the abandoned connections.

   o  When the connection is established (i.e., the three-way handshake
      has completed), an application request is sent, and the TCP
      connection is subsequently abandoned.  At this point, any state
      information kept by the attack tool is removed.

   o  The attacked server allocates TCP send buffers for transmitting
      the response to the client's request.  This causes the victim TCP
      to tie resources not only for the Transmission Control Block
      (TCB), but also for the application data that needs to be
      transferred.

   o  Once the application response is queued for transmission, the
      application closes the TCP connection, and thus TCP takes the
      responsibility to deliver the queued data.  Having the application
      close the connection has the benefit for the attacker that the
      application is not able to keep track of the number of TCP
      connections in use, and thus it is not able to enforce limits on
      the number of connections.

   o  The attacker repeats the above steps a large number of times, thus
      causing a large amount of system memory at the victim host to be
      tied to the abandoned connections.  When the system memory is
      exhausted, the victim host denies service to new connections, or
      possibly crashes.

   There are a number of factors that affect the effectiveness of this
   attack that are worth considering.  Firstly, while the attack is
   typically targeted at a service such as HTTP, the consequences of the
   attack are usually system-wide.  Secondly, depending on the size of
   the server's response, the underlying TCP connection may or may not

be closed: if the response is larger than the TCP send buffer size at
the server, the application will usually block in a call to write()
or send(), and would therefore not close the TCP connection, thus
allowing the application to enforce limits on the number of ongoing
connections.  Consequently, the attacker will usually try to elicit a
response that is equal to or slightly smaller than the send buffer of
the attacked TCP.  Thirdly, while [Shalunov, 2000] notes that one
visible effect of this attack is a large number of connections in the
FIN-WAIT-1 state, this will not usually be the case.  Given that the
attacker never acknowledges any segment other than the SYN/ACK
segment that is part of the three-way handshake, at the point in
which the attacked TCP tries to send the application's response the
congestion window (cwnd) will usually be 4*SMSS (four maximum-sized
segments).  If the application's response were larger than 4*SMSS,
even if the application had closed the connection, the FIN segment
would never be sent, and thus the connection would still remain in
the ESTABLISHED state (rather than transit to the FIN-WAIT-1 state).

## 7.1.2.  Countermeasures

The resource exhaustion attack described in Section 7.1.1 does not
necessarily differ from a legitimate high-load scenario, and
therefore is hard to mitigate without negatively affecting the
robustness of TCP.  However, complementary mitigations can still be
implemented to limit the impact of these attacks.

Enforcing limits on the number of connections with no user-space
controlling process

The considerations and recommendations in Section 5.3.2 for enforcing
limits on the number of connections with no user-space controlling
process are applicable to mitigate this vulnerability.

Enforcing per-user and per-process limits

While the Netkill attack is usually targeted at a service such as
HTTP, its impact is usually system-wide.  This is particularly
undesirable, as an attack against a single service might affect the
system as a whole (for example possibly precluding remote system
administration).

In order to avoid an attack against a single service from affecting
other services, we advise TCP implementations to enforce per-process
and per-user limits on maximum kernel memory that can be used at any
time.  Additionally, we recommend implementations to enforce per-
process and per-user limits on the number of existent TCP connections
at any time.

Limiting the number of ongoing connections at the application

The considerations and recommendations in Section 5.3.2 for enforcing
limits on the number of ongoing connections at the application are
applicable to mitigate this vulnerability.

Enabling applications to enforce limits on ongoing connections

As discussed in Section 6.1.1, the fact that the close() system call
marks the corresponding file descriptor as closed prevents the
application running on top of TCP from enforcing limits on the
corresponding connection.

While it is common practice for applications to terminate their
connections by means of the close() system call, it is possible for
an application to initiate the connection-termination phase without
closing the corresponding file descriptor (hence keeping control of
the connection).

In order to achieve this, an application performing an active close
(i.e., initiating the connection-termination phase) should replace
the call to close(sockfd) with the following code sequence:

o  A call to shutdown(sockfd, SHUT_WR), to close the sending
   direction of this connection

o  Successive calls to read(), until it returns 0, thus indicating
   that the remote TCP peer has finished sending data.

o  A call to setsockopt(sockfd, SOL_SOCKET, SO_LINGER, &l,
   sizeof(l)), where l is of type struct linger (with its members
   l.l_onoff=1 and l.l_linger=90).

o  A call to close(sockfd), to close the corresponding file
   descriptor.

The call to shutdown() (instead of close()) allows the application to
retain control of the underlying TCP connection while the connection
transitions through the FIN-WAIT-1 and FIN-WAIT-2 states.  However,
the application will not retain control of the connection while it
transitions through the CLOSING and TIME-WAIT states.  Nevertheless,
in these states TCP should not have any pending data to send to the
remote TCP peer or to be received by the application running on top
of it, and thus these states are less of a concern for this
particular vulnerability (Netkill).

It should be noted that, strictly speaking, close(sockfd) decrements
the reference count for the descriptor sockfd, and initiates the

connection termination phase only when the reference count reaches 0.
On the other hand, shutdown(sockfd, SHUT_WR) initiates the
connection-termination phase, regardless of the reference count for
the sockfd descriptor.  This should be taken into account when
performing the code replacement described above.  For example, it
would be a bug for two processes (e.g., parent and child) that share
a descriptor to both call shutdown(sockfd, SHUT_WR).

An application performing a passive close should replace the call to
close(sockfd) with the following code sequence:

o  A call to setsockopt(sockfd, SOL_SOCKET, SO_LINGER, &l,
   sizeof(l)), where l is of type struct linger (with its members
   l.l_onoff=1 and l.l_linger=90).

o  A call to close(sockfd), to close the corresponding file
   descriptor.

It is assumed that if the application is performing a passive close,
the application already detected that the remote TCP peer finished
sending data by means as a result of a call to read() returning 0.

In this scenario, the application will not retain control of the
underlying connection when it transitions through the LAST_ACK state.
However, in this state TCP should not have any pending data to send
to the remote TCP peer or to be received by the application running
on top of TCP, and thus this state is less of a concern for this
particular vulnerability (Netkill).

Limiting the number of simultaneous connections at firewalls

The considerations and recommendations in Section 5.3.2 for enforcing
limiting the number of simultaneous connections at firewalls are
applicable to mitigate this vulnerability.

Performing heuristics on ongoing TCP connections

Some heuristics could be performed on TCP connections that may
possibly help if scarce system requirements such as memory become
exhausted.  A number of parameters may be useful to perform such
heuristics.

In the case of the Netkill attack described in [Shalunov, 2000],
there are two parameters that are characteristic of a TCP being
attacked:

   o  A large amount of data queued in the TCP retransmission buffer
      (e.g., the socket send buffer).

   o  Only small amount of data has been successfully transferred to the
      remote peer.

   Clearly, these two parameters do not necessarily indicate an ongoing
   attack.  However, if exhaustion of the corresponding system resources
   was imminent, these two parameters (among others) could be used to
   perform heuristics when considering aborting ongoing connections.

   It should be noted that while an attacker could advertise a zero
   window to cause the target system to tie system memory to the TCP
   retransmission buffer, it is hard to perform any useful statistics
   from the advertised window.  While it is tempting to enforce a limit
   on the length of the persist state (see Section 3.7.2 of this
   document), an attacker could simply open the window (i.e., advertise
   a TCP window larger than zero) from time to time to prevent this
   enforced limit from causing his malicious connections to be aborted.

## 7.2.  TCP segment reassembly buffer

   TCP buffers out-of-order segments to more efficiently handle the
   occurrence of packet reordering and segment loss.  When out-of-order
   data are received, a "hole" momentarily exists in the data stream
   which must be filled before the received data can be delivered to the
   application making use of TCP's services.  This situation can be
   exploited by an attacker, which could intentionally create a hole in
   the data stream by sending a number of segments with a sequence
   number larger than the next sequence number expected (RCV.NXT) by the
   attacked TCP.  Thus, the attacked TCP would tie system memory to
   buffer the out-of-order segments, without being able to hand the
   received data to the corresponding application.

   If a large number of such connections were created, system memory
   could be exhausted, precluding the attacked TCP from servicing new
   connections and/or continue servicing TCP connections previously
   established.

   Fortunately, these attacks can be easily mitigated, at the expense of
   degrading the performance of possibly legitimate connections.  When
   out-of-order data is received, an Acknowledgement segment is sent
   with the next sequence number expected (RCV.NXT).  This means that
   receipt of the out-of-order data will not be actually acknowledged by
   the TCP's cumulative Acknowledgement Number.  As a result, a TCP is
   free to discard any data that have been received out-of-order,
   without affecting the reliability of the data transfer.  Given the
   performance implications of discarding out-of-order segments for

legitimate connections, this pruning policy should be applied only if
memory exhaustion is imminent.

As a result of discarding the out-of-order data, these data will need
to be unnecessarily retransmitted.  Additionally, a loss event will
be detected by the sending TCP, and thus the slow start phase of
TCP's congestion control will be entered, thus reducing the data
transfer rate of the connection.

It is interesting to note that this pruning policy could be applied
even if Selective Acknowledgements (SACK) (specified in RFC 2018
[Mathis et al, 1996]) are in use, as SACK provides only advisory
information, and does not preclude the receiving TCP from discarding
data that have been previously selectively-acknowledged by means of
TCP's SACK option, but not acknowledged by TCP's cumulative
Acknowledgement Number.

There are a number of ways in which the pruning policy could be
triggered.  For example, when out of order data are received, a timer
could be set, and the sequence number of the out-of-order data could
be recorded.  If the hole were filled before the timer expires, the
timer would be turned off.  However, if the timer expired before the
hole were filled, all the out-of-order segments of the corresponding
connection would be discarded.  This would be a proactive counter-
measure for attacks that aim at exhausting the receive buffers.

In addition, an implementation could incorporate reactive mechanisms
for more carefully controlling buffer allocation when some predefined
buffer allocation threshold was reached.  At such point, pruning
policies would be applied.

A number of mechanisms can aid in the process of freeing system
resources.  For example, a table of network prefixes corresponding to
the IP addresses of TCP peers that have ongoing TCP connections could
record the aggregate amount of out-of-order data currently buffered
for those connections.  When the pruning policy was triggered, TCP
connections with hosts that have network prefixes with large
aggregate out-of-order buffered data could be selected first for
pruning the out-of-order segments.

Alternatively, if TCP segments were de-multiplexed by means of a hash
table (as it is currently the case in many TCP implementations), a
counter could be held at each entry of the hash table that would
record the aggregate out-of-order data currently buffered for those
connections belonging to that hash table entry.  When the pruning
policy is triggered, the out-of-order data corresponding to those
connections linked by the hash table entry with largest amount of
aggregate out-of-order data could be pruned first.  It is important

that this hash is not computable by an attacker, as this would allow
him to maliciously cause the performance of specific connections to
be degraded.  That is, given a four-tuple that identifies a
connection, an attacker should not be able to compute the
corresponding hash value used by the target system to de-multiplex
incoming TCP segments to that connection.

Another variant of a resource exhaustion attack against TCP's segment
reassembly mechanism would target the data structures used to link
the different holes in a data stream.  For example, an attacker could
send a burst of 1 byte segments, leaving a one-byte hole between each
of the data bytes sent.  Depending on the data structures used for
holding and linking together each of the data segments, such an
attack might waste a large amount of system memory by exploiting the
overhead needed store and link together each of these one-byte
segments.

For example, if a linked-list is used for holding and linking each of
the data segments, each of the involved data structures could involve
one byte of kernel memory for storing the received data byte (the TCP
payload), plus 4 bytes (32 bits) for storing a pointer to the next
node in the linked-list.  Additionally, while such a data structure
would require only a few bytes of kernel memory, it could result in
the allocation of a whole memory page, thus consuming much more
memory than expected.

Therefore, implementations should enforce a limit on the number of
holes that are allowed in the received data stream at any given time.
When such a limit is reached, incoming TCP segments which would
create new holes would be silently dropped.  Measurements in
[Dharmapurikar and Paxson, 2005] indicate that in the vast majority
of TCP connections have at most a single hole at any given time.  A
limit of 16 holes for each connection would accommodate even most of
the very unusual cases in which there can be more than hole in the
data stream at a given time.

[US-CERT, 2004a] is a security advisory about a Denial of Service
vulnerability resulting from a TCP implementation that did not
enforce limits on the number of segments stored in the TCP reassembly
buffer.

Section 8 of this document describes the security implications of the
TCP segment reassembly algorithm.

## 7.3.  Automatic buffer tuning mechanisms

NOTE: THIS SECTION IS BEING EDITED.  PLEASE DISREGARD THE RFC2119-
LANGUAGE RECOMMENDATIONS.

### 7.3.1.  Automatic send-buffer tuning mechanisms

   A TCP implementing an automatic send-buffer tuning mechanism SHOULD
   enforce the following limit on the size of the send buffer of each
   TCP connection:

   send_buffer_size <= send_buffer_pool / (min_buffer_size * max_connections)

   where

   send_buffer_size:
      Maximum send buffer size to be used for this connection

   send_buffer_pool:
      Total amount of system memory meant for TCP send buffers

   min_buffer_size:
      Minimum send buffer size for each TCP connection

   max_connections:
      Maximum number of TCP connections this system is expected to
      handle at a time

   max_connections may be an artificial limit enforced by the system
   administrator specifically on the number of TCP connections, or may
   be derived from some other system limit (e.g., the maximum number of
   file descriptors)

   DISCUSSION:

      A number of TCP implementations incorporate automatic tuning
      mechanisms for the TCP send buffer size.  In most of them, the
      underlying idea is to set the send buffer to some multiple of the
      congestion window (cwnd).  This type of mechanism usually improves
      TCP's performance, by preventing the socket send buffer from
      becoming a bottleneck, while avoiding the need to simply
      overestimate the TCP send buffer size (i.e., make it arbitrarily
      large).  [Semke et al, 1998] discusses such an automatic buffer
      tuning mechanism.

      Unfortunately, automatic tuning mechanisms can be exploited by
      attackers to amplify the impact of other resource exhaustion
      attacks.  For example, an attacker could establish a TCP
      connection with a victim host, and cause the congestion window to
      be increased (either legitimately or illegitimately).  Once the
      congestion window (and hence the TCP send buffer) is increased, he
      could cause the corresponding system memory to be tied up by
      advertising a zero-byte TCP window (see Section 3.7) or simply not

acknowledging any data, thus amplifying the effect of resource
exhaustion attacks such as that discussed in Section 7.1.1.

When an automatic buffer tuning mechanism is implemented, a number
of countermeasures should be incorporated to prevent the mechanism
from being exploited to amplify other resource exhaustion attacks.

Firstly, appropriate policies should be applied to guarantee fair
use of the available system memory by each of the established TCP
connections.  Secondly, appropriate policies should be applied to
avoid existing TCP connections from consuming all system
resources, thus preventing service to new TCP connections.

Appendix A of [Semke et al, 1998] proposes an algorithm for the
fair share of the available system memory among the established
connections.  However, there are a number of limits that should be
enforced on the system memory assigned for the send buffer of each
connection.  Firstly, each connection should always be assigned
some minimum send buffer space that would enable TCP to perform at
an acceptable performance.  Secondly, some system memory should be
reserved for future connections, according to the maximum number
of concurrent TCP connections that are expected to be successfully
handled at any given time.

These limits preclude the automatic tuning algorithm from
assigning all the available memory buffers to ongoing connections,
thus preventing the establishment of new connections.

Even if these limits are enforced, an attacker could still create
a large number of TCP connections, each of them tying valuable
system resources.  Therefore, in scenarios in which most of the
system memory reserved for TCP send buffers is allocated to
ongoing connections, it may be necessary for TCP to enforce some
policy to free resources to either service more TCP connections,
or to be able to improve the performance of other existing
connections, by allocating more resources to them.

When needing to free memory in use for send buffers, particular
attention should be paid to TCP's that have a large amount of data
in the socket send buffer, and that at the same time fall into any
of these categories:


*  The remote TCP peer that has been advertising a small (possibly
   zero) window for a considerable period of time.

   *  There have been a large number of retransmissions of segments
      corresponding to the first few windows of data.

   *  Connections that fall into one of the previous categories, for
      which only a reduced amount of data have been successfully
      transferred to the peer TCP since the connection was
      established.

   Unfortunately, all these cases are valid scenarios for the TCP
   protocol, and thus aborting connections that fall in any of these
   categories has the potential of causing interoperability problems.
   However, in scenarios in which all system resources are allocated,
   it may make sense to free resources allocated to TCP connections
   which are tying a considerable amount of system resources and that
   have not made progress in a considerable period of time.

## 7.3.2.  Automatic receive-buffer tuning mechanism

   A number of TCP implementations include automatic tuning mechanisms
   for the receive buffer size.  These mechanisms aim at setting the
   socket buffer to a size that is large enough to avoid the TCP window
   from becoming a bottleneck that would limit TCP's throughput, without
   wasting system memory by over-sizing it.

   [Heffner, 2002] describes a mechanism for the automatic tuning of the
   socket receive buffer.  Basically, the mechanism aims at measuring
   the amount of data received during a RTT (Round-Trip Time), and
   setting the socket receive buffer to some multiple of that value.

   A TCP implementing an automatic receive-buffer tuning mechanism
   SHOULD enforce the following limit on the size of the receive buffer
   of each TCP connection:

   recv_buffer_size <= recv_buffer_pool / (min_buffer_size * max_connections)

   where:

   recv_buffer_size:
      Maximum receive buffer size to be used for this connection

   recv_buffer_pool:
      Total amount of system memory meant for TCP receive buffers

   min_buffer_size:
      Minimum receive buffer size for each TCP connection

   max_connections:
      Maximum number of TCP connections this system is expected to
      handle at a time

   max_connections may be an artificial limit enforced by the system
   administrator specifically on the number of TCP connections, or may
   be derived from some other system limit (e.g., the maximum number of
   file descriptors).

   DISCUSSION:

      Unfortunately, automatic tuning mechanisms for the socket receive
      buffer can be exploited to perform a resource exhaustion attack.
      An attacker willing to exploit the automatic buffer tuning
      mechanism would first establish a TCP connection with the victim
      host.  Subsequently, he would start a bulk data transfer to the
      victim host.  By carefully responding to the peer's TCP segments,
      the attacker could cause the peer TCP to measure a large data/RTT
      value, which would lead to the adoption of an unnecessarily large
      socket receive buffer.  For example, the attacker could
      optimistically send more data than those allowed by the TCP window
      advertised by the remote TCP.  Those extra data would cross in the
      network with the window updates sent by the remote TCP, and could
      lead the TCP receiver to measure a data/RTT twice as big as the
      real one.  Alternatively, if the TCP timestamp option (specified
      in RFC 1323 [Jacobson et al, 1992]) is used for RTT measurement,
      the attacker could lead the TCP receiver to measure a small RTT
      (and hence a large Data/RTT rate) by "optimistically" echoing
      timestamps that have not yet been received.

      Finally, once the TCP receiver is led to increase the size of its
      receive buffer, the attacker would transmit a large amount of
      data, filling the whole peer's receive buffer except for a few
      bytes at the beginning of the window (RCV.NXT).  This gap would
      prevent the peer application from reading the data queued by TCP,
      thus tying system memory to the received data segments until (if
      ever) the peer application times out.

      A number of limits should be enforced on the amount of system
      memory assigned to any given connection.  Firstly, each connection
      should always be assigned some minimum receive buffer space that
      would enable TCP to perform at a minimum acceptable performance.
      Additionally, some system memory should be reserved for future
      connections, according to the maximum number of concurrent TCP
      connections that are expected to be successfully handled at any
      given time.

These limits preclude the automatic tuning algorithm from
assigning all the available memory buffers to existing
connections, thus preventing the establishment of new connections.

It is interesting to note that a TCP sender will always try to
retransmit any data that have not been acknowledged by TCP's
cumulative acknowledgement.  Therefore, if memory exhaustion is
imminent, a system should consider freeing those memory buffers
used for TCP segments that were received out of order,
particularly when a given connection has been keeping a large
number of out-of-order segments in the receive buffer for a
considerable period of time.

It is worth noting that TCP Selective Acknowledgements (SACK) are
advisory, in the sense that a TCP that has SACKed (but not ACKed)
a block of data is free to discard that block, and expect the TCP
sender to retransmit them when the retransmission timer of the
peer TCP expires.


**8**.  **TCP segment reassembly algorithm**

**8.1**.  **Problems that arise from ambiguity in the reassembly process**

A security consideration that should be made for the TCP segment
reassembly algorithm is that of data stream consistency between the
host performing the TCP segment reassembly, and a Network Intrusion
Detection System (NIDS) being employed to monitor the host in
question.

In the event a TCP segment was unnecessarily retransmitted, or there
was packet duplication in any of the intervening networks, a TCP
might get more than one copy of the same data.  Also, as TCP segments
can be re-packetized when they are retransmitted, a given TCP segment
might partially overlap data already received in earlier segments.
In all these cases, the question arises about which of the copies of
the received data should be used when reassembling the data stream.
In legitimate and normal circumstances, all copies would be
identical, and the same data stream would be obtained regardless of
which copy of the data was used.  However, an attacker could
maliciously send overlapping segments containing different data, with
the intent of evading a Network Intrusion Detection Systems (NIDS),
which might reassemble the received TCP segments differently than the
monitored system.  [Ptacek and Newsham, 1998] provides a detailed
discussion of these issues.

As suggested in Section 3.9 of RFC 793 [RFC0793], if a TCP segment
arrives containing some data bytes that have already been received,

the first copy of those data should be used for reassembling the
application data stream.  It should be noted that while convergence
to this policy might prevent some cases of ambiguity in the
reassembly process, there are a number of other techniques that an
attacker could still exploit to evade a NIDS [CPNI, 2008].  These
techniques can generally be defeated if the NIDS is placed in-line
with the monitored system, thus allowing the NIDS to normalize the
network traffic or apply some other policy that could ensure
consistency between the result of the segment reassembly process
obtained by the monitored host and that obtained by the NIDS.

[CERT, 2003] and [CORE, 2003] are advisories about a heap buffer
overflow in a popular Network Intrusion Detection System resulting
from incorrect sequence number calculations in its TCP stream-
reassembly module.

## 9.  TCP Congestion Control

NOTE: THIS SECTION IS BEING EDITED.

TCP implements two algorithms, "slow start" and "congestion
avoidance", for controlling the rate at which data is transmitted on
a TCP connection [RFC5681].

### 9.1.  Congestion control with misbehaving receivers

[Savage et al, 1999] describes a number of ways in which TCP's
congestion control mechanisms can be exploited by a misbehaving TCP
receiver to obtain more than its fair share of bandwidth.  The
following subsections provide a brief discussion of these
vulnerabilities, along with the possible countermeasures.

### 9.1.1.  ACK division

Given that TCP updates cwnd based on the number of duplicate ACKs it
receives, rather than on the amount of data that each ACK is actually
acknowledging, a malicious TCP receiver could cause the TCP sender to
illegitimately increase its congestion window by acknowledging a data
segment with a number of separate Acknowledgements, each covering a
distinct piece of the received data segment.

              See Figure 7, in page 64 of the UK CPNI document.


                          ACK division attack

[Savage et al, 1999] describes two possible countermeasures for this
vulnerability.  One of them is to increment cwnd not by a full SMSS,

but proportionally to the amount of data being acknowledged by the
received ACK, similarly to the policy described in RFC 3465 [Allman,
2003].  Another alternative is to increase cwnd by one SMSS only when
a valid ACK covers the entire data segment sent.

## 9.1.2.  DupACK forgery

The second vulnerability discussed in [Savage et al, 1999] allows an
attacker to cause the TCP sender to illegitimately increase its
congestion window by forging a number of duplicate Acknowledgements
(DupACKs).  Figure 8 shows a sample scenario.  The first three
DupACKs trigger the Fast Recovery mechanism, while the rest of them
cause the congestion window at the TCP sender to be illegitimately
inflated.  Thus, the attacker is able to illegitimately cause the TCP
sender to increase its data transmission rate.

           See Figure 8, in page 65 of the UK CPNI document.

                        DupACK forgery attack

Fortunately, a number of sender-side heuristics can be implemented to
mitigate this vulnerability.  First, the TCP sender could keep track
of the number of outstanding segment (o_seg), and accept only up to
(o_seg -1) DupACKs.  Secondly, a TCP sender might, for example,
refuse to enter Fast Recovery multiple times in some period of time
(e.g., one RTT).

[Savage et al, 1999] also describes a modification to TCP to
implement a nonce protocol that would eliminate this vulnerability.
However, this would require modification of all implementations,
which makes this counter-measure hard to deploy.

## 9.1.3.  Optimistic ACKing

Another alternative for an attacker to exploit TCP's congestion
control mechanisms is to acknowledge data that has not yet been
received, thus causing the congestion window at the TCP sender to be
incremented faster than it should.

           See Figure 9, in page 66 of the UK CPNI document.

                       Optimistic ACKing attack

[Savage et al, 1999] describes a number of mitigations for this
vulnerability.  Firstly, it describes a countermeasure based on the
concept of "cumulative nonce", which would allow a receiver to prove
that it has received all the segments it is acknowledging.  However,
this countermeasure requires the introduction of two new fields to

the TCP header, thus requiring a modification to all the
communicating TCPs, makes this counter-measure hard to deploy.
Secondly, it describes a possible way to encode the nonce in a TCP
segment by carefully modifying its size.  While this countermeasure
could be easily deployed (as it is just sender side policy), we
believe that middle-boxes such as protocol-scrubbers might prevent
this counter-measure from working as expected.  Finally, it suggests
that a TCP sender might penalize a TCP receiver that acknowledges
data not yet sent by resetting the corresponding connection.  Here we
discourage the implementation of this policy, as it would provide an
attack vector for a TCP-based connection-reset attack, similar to
those described in Section 11.

[US-CERT, 2005a] is a vulnerability advisory about this issue.

## 9.2.  Blind DupACK triggering attacks against TCP

While all of the attacks discussed in [Savage et al, 1999] have the
goal of increasing the performance of the attacker's TCP connections,
TCP congestion control mechanisms can be exploited with a variety of
goals.

Firstly, if bursts of many duplicate-ACKs are sent to the "sending
TCP", the third duplicate-ACK will cause the "lost" segment to be
retransmitted, and each subsequent duplicate-ACK will cause cwnd to
be artificially inflated.  Thus, the "sending TCP" might end up
injecting more packets into the network than it really should, with
the potential of causing network congestion.  This is a potential
consequence of the "Duplicate-ACK spoofing attack" described in
[Savage et al, 1999].

Secondly, if bursts of three duplicate ACKs are sent to the TCP
sender, the attacked system would infer packet loss, and ssthresh and
cwnd would be reduced.  As noted in RFC 5681 [RFC5681], causing two
congestion control events back-to-back will often cut ssthresh and
cwnd to their minimum value of 2*SMSS, with the connection
immediately entering the slower-performing congestion avoidance
phase.  While it would not be attractive for an attacker to perform
this attack against one of his TCP connections, the attack might be
attractive when the TCP connection to be attacked is established
between two other parties.

It is usually assumed that in order for an off-path attacker to
perform attacks against a third-party TCP connection, he should be
able to guess a number of values, including a valid TCP Sequence
Number and a valid TCP Acknowledgement Number.  While this is true if
the attacker tries to "inject" valid packets into the connection by
himself, a feature of TCP can be exploited to fool one of the TCP

endpoints to transmit valid duplicate Acknowledgements on behalf of
the attacker, hence relieving the attacker of the hard task of
forging valid values for the Sequence Number and Acknowledgement
Number TCP header fields.

Section 3.9 of RFC 793 [RFC0793] describes the processing of incoming
TCP segments as a function of the connection state and the contents
of the various header fields of the received segment.  For
connections in the ESTABLISHED state, the first check that is
performed on incoming segments is that they contain "in window" data.
That is,

$$RCV.NXT <= SEG.SEQ <= RCV.NXT+RCV.WND, \text{ or}$$

$$RCV.NXT <= SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND$$

If a segment does not pass this check, it is dropped, and an
Acknowledgement is sent in response:

$$<SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>$$

The goal of this behavior is that, in the event data segments are
received by the TCP receiver, but all the corresponding
Acknowledgements are lost, when the TCP sender retransmits the
supposedly lost data, the TCP receiver will send an Acknowledgement
reflecting all the data received so far.  If "old" TCP segments were
silently dropped, the scenario just described would lead to a
"frozen" TCP connection, with the TCP sender retransmitting the data
for which it has not yet received an Acknowledgement, and the TCP
receiver silently ignoring these segments.  Additionally, it helps
TCP to detect half-open connections.

This feature implies that, provided the four-tuple that identifies a
given TCP connection is known or can be easily guessed, an attacker
could send a TCP segment with an "out of window" Sequence Number to
one of the endpoints of the TCP connection to cause it to send a
valid ACK to the other endpoint of the connection.  Figure 10
illustrates such a scenario.

          See Figure 10, in page 68 of the UK CPNI document.


                     Blind Dup-ACK forgery attack

As discussed in [Watson, 2004] and RFC 4953 [Touch, 2007], there are
a number of scenarios in which the four-tuple that identifies a TCP
connection is known or can be easily guessed.  In those scenarios, an
attacker could perform any of the "blind" attacks described in the

following subsections by exploiting the technique described above.

The following subsections describe blind DupACK-triggering attacks
that aim at either degrading the performance of an arbitrary
connection, or causing a TCP sender to illegitimately increase the
rate at which it transmits data, potentially leading to network
congestion.

## 9.2.1.  Blind throughput-reduction attack

As discussed in Section 9, when three duplicate Acknowledgements are
received, the congestion window is reduced to half the current amount
of outstanding data (FlightSize).  Additionally, the slow-start
threshold (ssthresh) is reduced to the same value, causing the
connection to enter the slower-performing congestion avoidance phase.
If two congestion-control events occur back to back, ssthresh and
cwnd will often be reduced to their minimum value of 2*SMSS.

An attacker could exploit the technique described in Section 9.2 to
cause the throughput of the attacked TCP connection to be reduced, by
eliciting three duplicate acknowledgements from the TCP receiver,
which would cause the TCP sender to reduce its congestion window.  In
principle, the attacker would need to send a burst of only three out-
of-window segments.  However, in case the TCP receiver implements an
acknowledgement policy such as "ACK every other segment", four out-
of-window segments might be needed.  The first segment would cause
the pending (delayed) Acknowledgement to be sent, and the next three
segments would elicit the actual duplicate Acknowledgements.

Figure 11 shows a time-line graph of a sample scenario.  The burst of
DupACKs (in green) elicited by the burst of out-of-window segments
(in red) sent by the attacker causes the TCP sender to retransmit the
missing segment (in blue) and enter the loss recovery phase.  Once a
segment that acknowledges new data is received by the TCP sender, the
loss recovery phase ends, and cwnd and ssthresh are set to half the
number of segments that were outstanding when the loss recovery phase
was entered.

           See Figure 11, in page 69 of the UK CPNI document.

           Blind throughput-reduction attack (time-line graph)

The graphic assumes that the TCP receiver sends an Acknowledgement
for every other data segment it receives, and that the TCP sender
implements Appropriate Byte Counting (specified in RFC 3465 [Allman,
2003]) on the received Acknowledgement segments.  However,
implementation of these policies is not required for the attack to
succeed.

### 9.2.2.  Blind flooding attack

As discussed in Section 9, when three duplicate Acknowledgements are
received, the "lost" segment is retransmitted, and the congestion
window is artificially inflated for each DupACK received, until the
loss recovery phase ends.  By sending a long burst of out-of-window
segments to the TCP receiver of the attacked connection, an attacker
could elicit a long burst of valid duplicate acknowledgements that
would illegitimately cause the TCP sender of the attacked TCP
connection to increase its data transmission rate.

Figure 12 shows a time-line graph for this attack.  The long burst of
DupACKs (in green) elicited by the long burst of out-of-window
segments (in red) sent by the attacker causes the TCP sender to enter
the loss recovery phase and illegitimately inflate the congestion
window, leading to an increase in the data transmission rate.  Once a
segment that acknowledges new data is received by the TCP sender, the
loss recovery phase ends, and the data transmission rate is reduced.

See Figure 12, in page 70 of the UK CPNI document.

Blind flooding attack (time-line graph)

### 9.2.3.  Difficulty in performing the attacks

In order to exploit the technique described in Section 9.2 of this
document, an attacker would need to know the four-tuple {IP Source
Address, TCP Source Port, IP Destination Address, TCP Destination
Port} that identifies the connection to be attacked.  As discussed by
[Watson, 2004] and RFC 4953 [Touch, 2007], there are a number of
scenarios in which these values may be known or easily guessed.

It is interesting to note that the attacks described in Section 9.2
of this document will typically require a much smaller number of
packets than other "blind" attacks against TCP, such as those
described in [Watson, 2004] and RFC 4953 [Touch, 2007], as the
technique discussed in Section 9.2 relieves the attacker from having
to guess valid TCP Sequence Numbers and a TCP Acknowledgement
numbers.

The attacks described in Section 9.2.1 and Section 9.2.2 of this
document require the attacker to forge the source address of the
packets it sends.  Therefore, if ingress/egress filtering is
performed by intermediate systems, the attacker's packets would not
get to the intended recipient, and thus the attack would not succeed.
However, we consider that ingress/egress filtering cannot be relied
upon as the first line of defense against these attacks.

Finally, it is worth noting that in order to successfully perform the blind attacks discussed in Section 9.2.1 and Section 9.2.2 of this document, the burst of out-of-sequence segments sent by the attacker should not be intermixed with valid data segments sent by the TCP sender, or else the Acknowledgement number of the illegitimately-elicited ACK segments would change, and the Acknowledgements would not be considered "Duplicate Acknowledgements" by the TCP sender. Tests performed in real networks seem to suggest that this requirement is not hard to fulfill, though.

### 9.2.4.  Modifications to TCP's loss recovery algorithms

There are a number of algorithms that augment TCP's loss recovery mechanism that have been suggested by TCP researchers and have been specified by the IETF in the RFC series.  This section describes a number of these algorithms, and discusses how their implementation affects (or not) the vulnerability of TCP to the attacks discussed in Section 9.2.1 and Section 9.2.2 of this document.

NewReno

RFC 3782 [Floyd et al, 2004] specifies the NewReno algorithm, which is meant to improve TCP's performance in the presence of multiple losses in a single window of data.  The implication of this algorithm with respect to the attacks discussed in the previous sections is that whenever either of the attacks is performed against a connection with a NewReno TCP sender, a full-window (or half a window) of data will be unnecessarily retransmitted.  This is particularly interesting in the case of the blind-flooding attack, as the attack would elicit even more packets from the TCP sender.

Whether a full-window or just half a window of data is retransmitted depends on the Acknowledgement policy at the TCP receiver.  If the TCP receiver sends an Acknowledgement (ACK) for every segment, a full-window of data will be retransmitted.  If the TCP receiver sends an Acknowledgement (ACK) for every other segment, then only half a window of data will be retransmitted.

Limited Transmit

RFC 3042 [Allman et al, 2001] proposes an enhancement to TCP to more effectively recover lost segments when a connection's congestion window is small, or when a large number of segments are lost in a single transmission window.  The "Limited Transmit" algorithm calls for sending a new data segment in response to each of the first two Duplicate Acknowledgements that arrive at the TCP sender.  This would provide two additional transmitted packets that may be useful for the attacker in the case of the blind flooding attack described in

Section 9.2.2 is performed.

SACK-based loss recovery

[I-D.ietf-tcpm-3517bis] specifies a conservative loss-recovery
algorithm that is based on the use of the selective acknowledgement
(SACK) TCP option.  The algorithm uses DupACKs as an indication of
congestion, as specified in RFC 2581 [RFC5681].  However, a
difference between this algorithm and the basic algorithm described
in RFC 2581 is that it clocks out segments only with the SACK
information included in the DupACKs.  That is, during the loss
recovery phase, segments will be injected in the network only if the
SACK information included in the received DupACKs indicates that one
or more segments have left the network.  As a result, those systems
that implement SACK-based loss recovery will not be vulnerable to the
blind flooding attack described in Section 9.2.2.  Additionally, as
[I-D.ietf-tcpm-3517bis] requires DupACKs to include new SACK
information (corresponding to data that has not yet been acknowledged
by TCP's cumulative Acknowledgement), systems that implement SACK-
based loss-recovery will not be vulnerable to the blind throughput-
reduction attack described in Section 9.2.1.

## 9.2.5.  Countermeasures

[draft-gont-tcpm-limiting-aow-segments-00.txt] proposes to rate-limit
the reaction to out-of-window segments.  This would mitigate the
attacks described earlier in this section.

## 9.3.  TCP Explicit Congestion Notification (ECN)

ECN (Explicit Congestion Notification) provides a mechanism for
intermediate systems to signal congestion to the communicating
endpoints that in some scenarios can be used as an alternative to
dropping packets.

RFC 3168 [Ramakrishnan et al, 2001] contains a detailed discussion of
the possible ways and scenarios in which ECN could be exploited by an
attacker.

RFC 3540 [Spring et al, 2003] specifies an improvement to ECN based
on nonces, that protects against accidental or malicious concealment
of marked packets from the TCP sender.  The specified mechanism
defines a "NS" ("Nonce Sum") field in the TCP header that makes use
of one bit from the Reserved field, and requires a modification in
both of the endpoints of a TCP connection to process this new field.
This mechanism is still in "Experimental" status, and since it might
suffer from the behavior of some middle-boxes such as firewalls or
packet-scrubbers, we defer a recommendation of this mechanism until

more experience is gained.

> There also is ongoing work in the research community and the IETF
> to define alternate semantics for the ECN field of the IP header
> (e.g., see [PCNWG, 2009]).

RFC 3168 [RFC3168] provides a very throrough security assessment of
ECN.  Among the possible mitigations, it describes the use of
"penalty boxes" which would act on flows that do not respond
appropriately to congestion indications.  Section 10 of RFC 3168
suggests that a first action taken at a penalty box for an ECN-
capable flow would be to switch to dropping packets (instead of
marking them), and, if the flow does not respond appropriately to the
congestion indication, the penalty box could reset the misbehaving
connection.  Here we discourage implementation of such a policy, as
it would create a vector for connection-reset attacks.  For example,
an attacker could forge TCP segments with the same four-tuple as the
targeted connection and cause them to transit the penalty box.  The
penalty box would first switch from marking to dropping packets.
However, the attacker would continue sending forged segments, at a
steady rate.  As a result, if the penalty box implemented such a
severe policy of resetting connections for flows that still do not
respond to end-to-end congestion control after switching from marking
to dropping, the attacked connection would be reset.

## 10.  TCP API

NOTE: THIS SECTION IS BEING EDITED.

Section 3.8 of RFC 793 [RFC0793] describes the minimum set of TCP
User Commands required of all TCP Implementations.  Most operating
systems provide an Application Programming Interface (API) that
allows applications to make use of the services provided by TCP.  One
of the most popular APIs is the Sockets API, originally introduced in
the BSD networking package [McKusick et al, 1996].

### 10.1.  Passive opens and binding sockets

When there is already a pending passive OPEN for some local port
number, TCP SHOULD NOT allow processes that do not belong to the same
user to "reuse" the local port for another passive OPEN.
Additionally, reuse of a local port SHOULD default to "off", and be
enabled only by an explicit command (e.g., the setsockopt() function
of the Sockets API).

DISCUSSION:

RFC 793 specifies the syntax of the "OPEN" command, which can be
used to perform both passive and active opens.  The syntax of this
command is as follows:

OPEN (local port, foreign socket, active/passive [, timeout] [,
precedence] [, security/compartment] [, options]) -> local
connection name

When this command is used to perform a passive open (i.e., the
active/passive flag is set to passive), the foreign socket
parameter may be either fully-specified (to wait for a particular
connection) or unspecified (to wait for any call).

As discussed in Section 2.7 of RFC 793 [RFC0793], if there are
several passive OPENs with the same local socket (recorded in the
corresponding TCB), an incoming connection will be matched to the
TCB with the more specific foreign socket.  This means that when
the foreign socket of a passive OPEN matches that of the incoming
connection request, that passive OPEN takes precedence over those
passive OPENs with an unspecified foreign socket.

Popular implementations such as the Sockets API let the user
specify the local socket as fully-specified {local IP address,
local TCP port} pair, or as just the local TCP port (leaving the
local IP address unspecified).  In the former case, only those
connection requests sent to {local port, local IP address} will be
accepted.  In the latter case, connection requests sent to any of
the system's IP addresses will be accepted.  In a similar fashion
to the generic API described in Section 2.7 of RFC 793, if there
is a pending passive OPEN with a fully-specified local socket that
matches that for which a connection establishment request has been
received, that local socket will take precedence over those which
have left the local IP address unspecified.  The implication of
this is that an attacker could "steal" incoming connection
requests meant for a local application by performing a passive
OPEN that is more specific than that performed by the legitimate
application.

## 10.2.  Active opens and binding sockets

TCP SHOULD NOT allow port numbers that have been allocated for a TCP
that is the LISTEN or CLOSED states to be specified as the "local
port" argument of the "OPEN" command.

An implementation MAY relax the aforementioned restriction when the
process or system user requesting allocation of such a port number is
the same that the process or system user controlling the TCP in the
CLOSED or LISTEN states with the same port number.

DISCUSSION:

As discussed in Section 10.1, the "OPEN" command specified in
Section 3.8 of RFC 793 [RFC0793] can be used to perform active
opens.  In case of active opens, the parameter "local port" will
contain a so-called "ephemeral port".  While the only requirement
for such an ephemeral port is that the resulting connection-id is
unique, port numbers that are currently in use by a TCP in the
LISTEN state should not be allowed for use as ephemeral ports.  If
this rule is not complied, an attacker could potentially "steal"
an incoming connection to a local server application by issuing a
connection request to the victim client at roughly the same time
the client tries to connect to the victim server application.  If
the SYN segment corresponding to the attacker's connection request
and the SYN segment corresponding to the victim client "cross each
other in the network", and provided the attacker is able to know
or guess the ephemeral port used by the client, a TCP simultaneous
open scenario would take place, and the incoming connection
request sent by the client would be matched with the attacker's
socket rather than with the victim server application's socket.

As already noted, in order for this attack to succeed, the
attacker should be able to guess or know (in advance) the
ephemeral port selected by the victim client, and be able to know
the right moment to issue a connection request to the victim
client.  While in many scenarios this may prove to be a difficult
task, some factors such as an inadequate ephemeral port selection
policy at the victim client could make this attack feasible.

It should be noted that most applications based on popular
implementations of TCP API (such as the Sockets API) perform
"passive opens" in three steps.  Firstly, the application obtains
a file descriptor to be used for inter-process communication
(e.g., by issuing a socket() call).  Secondly, the application
binds the file descriptor to a local TCP port number (e.g., by
issuing a bind() call), thus creating a TCP in the fictional
CLOSED state.  Thirdly, the aforementioned TCP is put in the
LISTEN state (e.g., by issuing a listen() call).  As a result,
with such an implementation of the TCP API, even if port numbers
in use for TCPs in the LISTEN state were not allowed for use as
ephemeral ports, there is a window of time between the second and
the third steps in which an attacker could be allowed to select a
port number that would be later used for listening to incoming
connections.  Therefore, these implementations of the TCP API
should enforce a stricter requirement for the allocation of port
numbers: port numbers that are in use by a TCP in the LISTEN or
CLOSED states should not be allowed for allocation as ephemeral
ports.

An implementation might choose to relax the aforementioned
restriction when the process or system user requesting allocation
of such a port number is the same that the process or system user
controlling the TCP in the CLOSED or LISTEN states with the same
port number.

## 11. Blind in-window attacks

NOTE: THIS SECTION IS BEING EDITED.

In the last few years awareness has been raised about a number of
"blind" attacks that can be performed against TCP by forging TCP
segments that fall within the receive window [NISCC, 2004] [Watson,
2004].

The term "blind" refers to the fact that the attacker does not have
access to the packets that belong to the attacked connection.

The effects of these attacks range from connection resets to data
injection.  While these attacks were known in the research community,
they were generally considered unfeasible.  However, increases in
bandwidth availability and the use of larger TCP windows raised
concerns in the community.  The following subsections discuss a
number of forgery attacks against TCP, along with the possible
countermeasures to mitigate their impact.

### 11.1. Blind TCP-based connection-reset attacks

Blind connection-reset attacks have the goal of causing a TCP
connection maintained between two TCP endpoints to be aborted.  The
level of damage that the attack may cause usually depends on the
application running on top of TCP, with the more vulnerable
applications being those that rely on long-lived TCP connections.

An interesting case of such applications is BGP [Rekhter et al,
2006], in which a connection-reset usually results in the
corresponding entries of the routing table being flushed.

There are a variety of vectors for performing TCP-based connection-
reset attacks against TCP.  [Watson, 2004] and [NISCC, 2004] raised
awareness about connection-reset attacks that exploit the RST flag of
TCP segments.  [Ramaiah et al, 2008] noted that carefully crafted SYN
segments could also be used to perform connection-reset attacks.
This document describes yet two previously undocumented vectors for
performing connection-reset attacks: the Precedence field of IP
packets that encapsulate TCP segments, and illegal TCP options.

**11.1.1**.  **RST flag**

   The RST flag signals a TCP peer that the connection should be
   aborted.  In contrast with the FIN handshake (which gracefully
   terminates a TCP connection), an RST segment causes the connection to
   be abnormally closed.

   As stated in Section 3.4 of RFC 793 [RFC0793], all reset segments are
   validated by checking their Sequence Numbers, with the Sequence
   Number considered valid if it is within the receive window.  In the
   SYN-SENT state, however, an RST is valid if the Acknowledgement
   Number acknowledges the SYN segment that supposedly elicited the
   reset.

   [RFC5961] proposes a modification to TCP's transition diagram to
   address this attack vector.  The counter-measure is a combination of
   enforcing a more strict validation check on the sequence number of
   reset segments, and the addition of a "challenge" mechanism.

      We note that we are aware of patent claims on this counter-
      measure, and suggest vendors to research the consequences of the
      possible patents that may apply.

   [US-CERT, 2003a] is an advisory of a firewall system that was found
   particularly vulnerable to resets attack because of not validating
   the TCP Sequence Number of RST segments.  Clearly, all TCPs
   (including those in middle-boxes) should validate RST segments as
   discussed in this section.

**11.1.2**.  **SYN flag**

   Section 3.9 (page 71) of RFC 793 [RFC0793] states that if a SYN
   segment is received with a valid (i.e., "in window") Sequence Number,
   an RST segment should be sent in response, and the connection should
   be aborted.  This could be leveraged to perform a blind connection-
   reset attack.  [RFC5961] proposes a change in TCP's state diagram to
   mitigate this attack vector.

**11.1.3**.  **Security/Compartment**

   Section 3.9 (page 71) of RFC 793 [RFC0793] states that if the IP
   security/compartment of an incoming segment does not exactly match
   the security/compartment in the TCB, a RST segment should be sent,
   and the connection should be aborted.  This certainly provides
   another attack vector for performing connection-reset attacks, as an
   attacker could forge TCP segments with a security/compartment that is
   different from that recorded in the corresponding TCB and, as a
   result, the attacked connection would be reset.

[draft-gont-tcpm-tcp-seccomp-prec-00.txt] aims to update RFC 793 such
that this issue is eliminated.

### 11.1.4.  Precedence

Section 3.9 (page 71) of RFC 793 [RFC0793] states that if the IP
precedence of an incoming segment does not exactly match the
precedence in the TCB, a RST segment should be sent, and the
connection should be aborted.  This certainly provides another attack
vector for performing connection-reset attacks, as an attacker could
forge TCP segments with a precedence that is different from that
recorded in the corresponding TCB and, as a result, the attacked
connection would be reset.

[draft-gont-tcpm-tcp-seccomp-prec-00.txt] aims to update RFC 793 such
that this issue is eliminated.

### 11.1.5.  Illegal options

Section 4.2.2.5 of RFC 1122 [RFC1122] discusses the processing of TCP
options.  It states that TCP should be prepared to handle an illegal
option length (e.g., zero) without crashing, and suggests handling
such illegal options by resetting the corresponding connection and
logging the reason.  However, this suggested behavior could be
exploited to perform connection-reset attacks.

[draft-gont-tcpm-tcp-illegal-option-lengths-00] aims at formally
updating RFC 1122, such that this issue is eliminated.

### 11.2.  Blind data-injection attacks

An attacker could try to inject data in the stream of data being
transferred on the connection.  As with the other attacks described
in Section 11 of this document, in order to perform a blind data
injection attack the attacker would need to know or guess the four-
tuple that identifies the TCP connection to be attacked.
Additionally, he should be able to guess a valid ("in window") TCP
Sequence Number, and a valid Acknowledgement Number.

As discussed in Section 3.4 of this document, [Ramaiah et al, 2008]
proposes to enforce a more strict check on the Acknowledgement Number
of incoming segments than that specified in RFC 793 [RFC0793].

Implementation of the proposed check requires more packets on the
side of the attacker to successfully perform a blind data-injection
attack.  However, it should be noted that applications concerned with
any of the attacks discussed in Section 11 of this document should
make use of proper authentication techniques, such as those specified

for IPsec in RFC 4301 [Kent and Seo, 2005].


## 12. Information leaking

NOTE: THIS SECTION IS BEING EDITED.

### 12.1. Remote Operating System detection via TCP/IP stack fingerprinting

Clearly, remote Operating System (OS) detection is a useful tool for
attackers.  Tools such as nmap [Fyodor, 2006b] can usually detect the
operating system type and version of a remote system with an
amazingly accurate precision.  This information can in turn be used
by attackers to tailor their exploits to the identified operating
system type and version.

Evasion of OS fingerprinting can prove to be a very difficult task.
Most systems make use of a variety of protocols, each of which have a
large number of parameters that can be set to arbitrary values.
Thus, information on the operating system may be obtained from a
number of sources ranging from application banners to more obscure
parameters such as TCP's retransmission timer.

Nmap [Fyodor, 2006b] is probably the most popular tool for remote OS
detection via active TCP/IP stack fingerprinting. p0f [Zalewski,
2006a], on the other hand, is a tool for performing remote OS
detection via passive TCP/IP stack fingerprinting.  SinFP [SinFP,
2006] can perform both active and passive fingerprinting.  Finally,
TBIT [TBIT, 2001] is a TCP fingerprinting tool that aims at
characterizing the behavior of a remote TCP peer based on active
probes, and which has been widely used in the research community.

TBIT [TBIT, 2001] implements a number of tests not present in other
tools, such as characterizing the behavior of a TCP peer with respect
to TCP congestion control.

[Fyodor, 1998] and [Fyodor, 2006a] are classic papers on the subject.
[Miller, 2006] and [Smith and Grundl, 2002] provide an introduction
to passive TCP/IP stack fingerprinting.  [Smart et al, 2000] and
[Beck, 2001] discuss some techniques for evading OS detection through
TCP/IP stack fingerprinting.

The following subsections discuss TCP-based techniques for remote OS
detection via and, where possible, propose ways to mitigate them.

### 12.1.1.  FIN probe

TCP MUST silently drop TCP any segments received for a connection in
the LISTEN state that do not have the SYN, RST, or ACK flags set.  In
the rest of the cases, the processing rules in RFC 793 MUST be
applied.

DISCUSSION:

   The attacker sends a FIN (or any packet without the SYN or the ACK
   flags set) to an open port.  RFC 793 [RFC0793] leaves the reaction
   to such segments unspecified.  As a result, some implementations
   silently drop the received segment, while others respond with a
   RST.

### 12.1.2.  Bogus flag test

TCP MUST ignore any flags not supported, and MUST NOT reflect them if
a TCP segment is sent in response to the one just received.

DISCUSSION:

   The attacker sends a TCP segment setting at least one bit of the
   Reserved field.  Some implementations ignore this field, while
   others reset the corresponding connection or reflect the field in
   the TCP segment sent in response.

### 12.1.3.  TCP ISN sampling

The attacker samples a number of Initial Sequence Numbers by sending
a number of connection requests.  Many TCP implementations differ on
the ISN generator they implement, thus allowing the correlation of
ISN generation algorithm to the operating system type and version.

This document advises implementing an ISN generator that follows the
behavior described in RFC 1948 [Bellovin, 1996].  However, it should
be noted that even if all TCP implementations generated their ISNs as
proposed in RFC 1948, there is still a number of implementation
details that are left unspecified, which would allow remote OS
fingerprinting by means of ISN sampling.  For example, the time-
dependent parameter of the hash could have a different frequency in
different TCP implementations.

### 12.1.4.  TCP initial window

Many TCP implementations differ on the initial TCP window they use.
There are a number of factors that should be considered when
selecting the TCP window to be used for a given system.  A number of

implementations that use static windows (i.e., no automatic buffer
tuning mechanisms are implemented) default to a window of around 32
KB, which seems sensible for the general case.  On the other hand, a
window of 4 KB seems to be common practice for connections servicing
critical applications such as BGP.  It is clear that the window size
is a tradeoff among a number of considerations.  Section 3.7
discusses some of the considerations that should be made when
selecting the window size for a TCP connection.

If automatic tuning mechanisms are implemented, we suggest the
initial window to be at least 4 * RMSS segments.  We note that a
remote OS fingerprinting tool could still sample the advertised TCP
window, trying to correlate the advertised window with the potential
automatic buffer tuning algorithm and Operating System.

## 12.1.5.  RST sampling

If an RST must be sent in response to an incoming segment, then if
the ACK bit of an incoming TCP segment is off, a Sequence Number of
zero MUST be used in the RST segment sent in response.  That is,

           <SEQ=0><ACK=SEG.SEQ+SEG.LEN><CTL=RST, ACK>

It should be noted that the SEG.LEN value used for the
Acknowledgement Number MUST be incremented once for each flag set in
the original segment that makes use of a byte of the sequence number
space.  That is, if only one of the SYN or FIN flags were set in the
received segment, the Acknowledgement Number of the response should
be set to SEG.SEQ+SEG.LEN+1.  If both the SYN and FIN flags were set
in the received segment, the Acknowledgement Number should be set to
SEG.SEQ+SEG.LEN+2.

We also RECOMMEND that TCP sets ACK bit (and the Acknowledgement
Number) in all outgoing RST segments, as it allows for additional
validation checks to be enforced at the system receiving the segment.

DISCUSSION:

   [Fyodor, 1998] reports that many implementations differ in the
   Acknowledgement Number they use in response to segments received
   for connections in the CLOSED state.  In particular, these
   implementations differ in the way they construct the RST segment
   that is sent in response to those TCP segments received for
   connections in the CLOSED state.

   RFC 793 [RFC0793] describes (in pages 36-37) how RST segments are
   to be generated.  According to this RFC, the ACK bit (and the
   Acknowledgment Number) is set in a RST only if the incoming

segment that elicited the RST did not have the ACK bit set (and
thus the Sequence Number of the outgoing RST segment must be set
to zero).  However, we recommend TCP implementations to set the
ACK bit (and the Acknowledgement Number) in all outgoing RST
segments, as it allows for additional validation checks to be
enforced at the system receiving the segment.

## 12.1.6.  TCP options

Different implementations differ in the TCP options they enable by
default.  Additionally, they differ in the actual contents of the
options, and in the order in which the options are included in a TCP
segment.  There is currently no recommendation on the order in which
to include TCP options in TCP segments.

## 12.1.7.  Retransmission Timeout (RTO) sampling

TCP uses a retransmission timer for retransmitting data in the
absence of any feedback from the remote data receiver.  The duration
of this timer is referred to as "retransmission timeout" (RTO).  RFC
2988 [Paxson and Allman, 2000] specifies the algorithm for computing
the TCP retransmission timeout (RTO).

The algorithm allows the use of clocks of different granularities, to
accommodate the different granularities used by the existing
implementations.  Thus, the difference in the resulting RTO can be
used for remote OS fingerprinting.  [Veysset et al, 2002] describes
how to perform remote OS fingerprinting by sampling and analyzing the
RTO of the target system.  However, this fingerprinting technique has
at least the following drawbacks:

o  It is usually much slower than other fingerprinting techniques, as
   it may require considerable time to sample the RTO of a given
   target.

o  It is less reliable than other fingerprinting techniques, as
   latency and packet loss can lead to bogus results.

While in principle it would be possible to defeat this fingerprinting
technique (e.g., by obfuscating the granularity of the clock used for
computing the RTO), we consider that a more important step to defeat
remote OS detection is for implementations to address the more
effective fingerprinting techniques described in Sections 12.1.1
through 12.1.7 of this document.

**12.2**.  **System uptime detection**

   The "uptime" of a system may prove to be valuable information to an
   attacker.  For example, it might reveal the last time a security
   patch was applied.  Information about system uptime is usually leaked
   by TCP header fields or options that are (or may be) time-dependent,
   and are usually initialized to zero when the system is bootstrapped.
   As a result, if the attacker knows the frequency with which the
   corresponding parameter or header field is incremented, and is able
   to sample the current value of that parameter or header field, the
   system uptime will be easily obtained.  Two fields that can
   potentially reveal the system uptime is the Sequence Number field of
   a SYN or SYN/ACK segment (i.e., when it contains an ISN) and the
   TSval field of the timestamp option.  Section 3.3.1 of this document
   discusses the generation of TCP Initial Sequence Numbers.  Section
   4.7.1 of this document discusses the generation of TCP timestamps.


**13**.  **Covert channels**

   As virtually every communications protocol, TCP can be exploited to
   establish covert channels.  While an exhaustive discussion of covert
   channels is out of the scope of this document, for completeness of
   the document we simply note that it is possible for a (probably
   malicious) user to establish a covert channel by means of TCP, such
   that data can be surreptitiously passed to a remote system, probably
   unnoticed by a monitoring system, and with the possibility of
   concealing the location of the source system.

   In most cases, covert channels based on manipulation of TCP fields
   can be eliminated by protocol scrubbers and other middle-boxes.  On
   the other hand, "timing channels" may prove to be more difficult to
   eliminate.

   [Rowland, 1996] contains a discussion of covert channels in the
   TCP/IP protocol suite, with some TCP-based examples.  [Giffin et al,
   2002] describes the use of TCP timestamps for the establishment of
   covert channels.  [Zander, 2008] contains an extensive bibliography
   of papers on covert channels, and a list of freely-available tools
   that implement covert channels with the TCP/IP protocol suite.


**14**.  **TCP Port scanning**

   NOTE: THIS SECTION IS BEING EDITED.

   TCP port scanning aims at identifying TCP port numbers on which there
   is a process listening for incoming connections.  That is, it aims at

identifying TCPs at the target system that are in the LISTEN state.
The following subsections describe different TCP port scanning
techniques that have been implemented in freely-available tools.
These subsections focus only on those port scanning techniques that
exploit features of TCP itself, and not of other communication
protocols.

For example, the following subsections do not discuss the
exploitation of application protocols (such as FTP) or the
exploitation of features of underlying protocols (such as the IP
Identification field) for port-scanning purposes.

## 14.1.  Traditional connect() scan

The most trivial scanning technique consists in trying to perform the
TCP three-way handshake with each of the port numbers at the target
system (e.g. by issuing a call to the connect() function of the
Sockets API).  The three-way handshake will complete for port numbers
that are "open", but will fail for those port numbers that are
"closed".

As this port-scanning technique can be implemented by issuing a call
to the connect() function of the Sockets API that normal applications
use, it does not require the attacker to have superuser privileges.
The downside of this port-scanning technique is that it is less
efficient than other scanning methods (e.g., the "SYN scan" described
in Section 14.2), and that it can be easily logged by the target
system.

## 14.2.  SYN scan

The SYN scan was introduced as a "stealth" port-scanning technique.
It aims at avoiding the target system from logging the port scan by
not completing the TCP three-way handshake.  When a SYN/ACK segment
is received in response to the initial SYN segment, the system
performing the port scan will respond with an RST segment, thus
preventing the three-way handshake from completing.  While this port-
scanning technique is harder to detect and log than the traditional
connect() scan described in Section 14.1, most current NIDS (Network
Intrusion Detection Systems) can detect and log it.

SYN scans are sometimes mistakenly reported as "SYN flood" attacks by
NIDS, though.

The main advantage of this port scanning technique is that it is much
more efficient than the traditional connect() scan.

In order to implement this port-scanning technique, port-scanning

tools usually bypass the TCP API, and forge the SYN segments they
send (e.g., by using raw sockets).  This typically requires the
attacker to have superuser privileges to be able to run the port-
scanning tool.

14.3.  FIN, NULL, and XMAS scans

TCP SHOULD respond with an RST when a TCP segment is received for a
connection in the LISTEN state, and the incoming segment has neither
the SYN bit nor the RST bit set.

DISCUSSION:

   RFC 793 [RFC0793] states, in page 65, that an incoming segment
   that does not have the RST bit set and that is received for a
   connection in the fictional state CLOSED causes an RST to be sent
   in response.  Pages 65-66 of RFC 793 describes the processing of
   incoming segments for connections in the state LISTEN, and
   implicitly states that an incoming segment that does not have the
   ACK bit set (and is not a SYN or an RST) should be silently
   dropped.

   As a result, an attacker can exploit this situation to perform a
   port scan by sending TCP segments that do not have the ACK bit set
   to the target system.  When a port is "open" (i.e., there is a TCP
   in the LISTEN state on the corresponding port), the target system
   will respond with an RST segment.  On the other hand, if the port
   is "closed" (i.e., there is a TCP in the fictional state CLOSED)
   the attacker will not get any response from the target system.

   Since the only requirement for exploiting this port scanning
   vector is that the probe segments must not have the ACK bit set,
   there are a number of different TCP control-bits combinations that
   can be used for the probe segments.

   When the probe segment sent to the target system is a TCP segment
   that has only the FIN bit set, the scanning technique is usually
   referred to as a "FIN scan".  When the probe packet is a TCP
   segment that does not have any of the control bits set, the
   scanning technique is usually known as a "NULL scan".  Finally,
   when the probe packet sent to the target system has only the FIN,
   PSH, and the URG bits set, the port-scanning technique is known as
   a "XMAS scan".

   It should be clear that while the aforementioned control-bits
   combinations are the most popular ones, other combinations could
   be used to exploit this port-scanning vector.  For example, the
   CWR, ECE, and/or any of the Reserved bits could be set in the

probe segments.

The advantage of this port-scanning technique is that in can
bypass some stateless firewalls.  However, the downside is that a
number of implementations do not comply strictly with RFC 793
[RFC0793], and thus always respond to the probe segments with an
RST, regardless of whether the port is open or closed.

This port-scanning vector can be easily defeated as rby responding
with an RST when a TCP segment is received for a connection in the
LISTEN state, and the incoming segment has neither the SYN bit nor
the RST bit set.

## 14.4.  Maimon scan

If a TCP that is in the CLOSED or LISTEN states receives a TCP
segment with both the FIN and ACK bits set, it MUST respond with a
RST.

DISCUSSION:

This port scanning technique was introduced in [Maimon, 1996] with
the name "StealthScan" (method #1), and was later incorporated
into the nmap tool [Fyodor, 2006b] as the "Maimon scan".

This port scanning technique employs TCP segments that have both
the FIN and ACK bits sets as the probe segments.  While according
to RFC 793 [RFC0793] these segments should elicit an RST
regardless of whether the corresponding port is open or closed, a
programming flaw found in a number of TCP implementations has
caused some systems to silently drop the probe segment if the
corresponding port was open (i.e., there was a TCP in the LISTEN
state), and respond with an RST only if the port was closed.

Therefore, an RST would indicate that the scanned port is closed,
while the absence of a response from the target system would
indicate that the scanned port is open.

While this bug has not been found in current implementations of
TCP, it might still be present in some legacy systems.

## 14.5.  Window scan

When sending an RST segment, TCP SHOULD set the Window field to zero.

DISCUSSION:

This port-scanning technique employs ACK segments as the probe
packets.  ACK segments will elicit an RST from the target system
regardless of whether the corresponding TCP port is open or
closed.  However, as described in [Maimon, 1996], some systems set
the Window field of the RST segments with different values
depending on whether the corresponding TCP port is open or closed.
These systems set the Window field of their RST segments to zero
when the corresponding TCP port is closed, and set the Window
field to a non-zero value when the corresponding TCP port is open.

As a result, an attacker could exploit this situation for
performing a port scan by sending ACK segments to the target
system, and examining the Window field of the RST segments that
his probe segments elicit.

In order to defeat this port-scanning technique, we recommend TCP
implementations to set the Window field to zero in all the RST
segments they send.  Most popular implementations of TCP already
implement this policy.

## 14.6.  ACK scan

The so-called "ACK scan" is not really a port-scanning technique
(i.e., it does not aim at determining whether a specific port is open
or closed), but rather aims at determining whether some intermediate
system is filtering TCP segments sent to that specific port number.

The probe packet is a TCP segment with the ACK bit set which,
according to RFC 793 [RFC0793] should elicit an RST from the target
system regardless of whether the corresponding TCP port is open or
closed.  If no response is received from the target system, it is
assumed that some intermediate system is filtering the probe packets
sent to the target system.

It should be noted that this "port scanning" techniques exploits
basic TCP processing rules, and therefore cannot be defeated at an
end-system.

## 15.  Processing of ICMP error messages by TCP

[RFC5927] analyzes a number of vulnerabilities based on crafted ICMP
messages, along with possible counter-measures.

## 16.  TCP interaction with the Internet Protocol (IP)

16.1.  TCP-based traceroute

   The traceroute tool is used to identify the intermediate systems the
   local system and the destination system.  It is usually implemented
   by sending "probe" packets with increasing IP Time to Live values
   (starting from 0), without maintaining any state with the final
   destination.

   Some traceroute implementations use ICMP "echo request" messages as
   the probe packets, while others use UDP packets or TCP SYN segments.

   In some cases, the state-less nature of the traceroute tool may
   prevent it from working correctly across stateful devices such as
   Network Address Translators (NATs) or firewalls.

   In order to by-pass this limitation, an attacker could establish a
   TCP connection with the destination system, and start sending TCP
   segments on that connection with increasing IP Time to Live values
   (starting from 0) [Zalewski, 2007] [Zalewski, 2008].  Provided ICMP
   error messages are not blocked by any intermediate system, an
   attacker could exploit this technique to map the network topology
   behind the aforementioned stateful devices in scenarios in which he
   could not have achieved this goal using the traditional traceroute
   tool.

   NATs [Srisuresh and Egevang, 2001] and other middle-boxes could
   defeat this network-mapping technique by overwriting the Time to Live
   of the packets they forward to the internal network.  For example,
   they could overwrite the Time to Live of all packets being forwarded
   to an internal network with a value such as 128.  We strongly
   recommend against overwriting the IP Time to Live field with the
   value 255 or other similar large values, as this could allow an
   attacker to bypass the protection provided by the Generalized TTL
   Security Mechanism (GTSM) described in RFC 5087 [Gill et al, 2007].

   [Gont and Srisuresh, 2008] discusses the security implications of
   NATs, and proposes mitigations for this and other issues.

16.2.  Blind TCP data injection through fragmented IP traffic

   As discussed in Section 11.2, TCP data injection attacks usually
   require an attacker to guess or know a number of parameters related
   with the target TCP connection, such as the connection-id {Source
   Address, Source Port, Destination Address, Destination Port}, the TCP
   Sequence Number, and the TCP Acknowledgement Number.  Provided these
   values are obfuscated as recommended in this document, the chances of
   an off-path attacker of successfully performing a data injection
   attack against a TCP connection are fairly low for many of the most

common scenarios.

As discussed in this document, randomization of the values contained
in different TCP header fields is not a replacement for cryptographic
methods for protecting a TCP connection, such as IPsec (specified in
RFC 4301 [Kent and Seo, 2005]).

However, [Zalewski, 2003b] describes a possible vector for performing
a TCP data injection attack that does not require the attacker to
guess or know the aforementioned TCP connection parameters, and could
therefore be successfully exploited in some scenarios with less
effort than that required to exploit the more traditional data-
injection attack vectors.

The attack vector works as follows.  When one system is transferring
information to a remote peer by means of TCP, and the resulting
packet gets fragmented, the first fragment will usually contain the
entire TCP header which, together with the IP header, includes all
the connection parameters that an attacker would need to guess or
know to successfully perform a data injection attack against TCP.  If
an attacker were able to forge all the fragments other than the first
one, his forged fragments could be reassembled together with the
legitimate first fragment, and thus he would be relieved from the
hard task of guessing or knowing connection parameters such as the
TCP Sequence Number and the TCP Acknowledgement Number.

In order to successfully exploit this attack vector, the attacker
should be able to guess or know both of the IP addresses involved in
the target TCP connection, the IP Identification value used for the
specific packet he is targeting, and the TCP Checksum of that target
packet.  While it would seem that these values are hard to guess, in
some specific scenarios, and with some security-unwise implementation
approaches for the TCP and IP protocols, these values may be feasible
to guess or know.  For example, if the sending system uses
predictable IP Identification values, the attacker could simply
perform a brute force attack, trying each of the possible
combinations for the TCP Checksum field.  In more specific scenarios,
the attacker could have more detailed knowledge about the data being
transferred over the target TCP connection, which might allow him to
predict the TCP Checksum of the target packet.  For example, if both
of the involved TCP peers used predictable values for the TCP
Sequence Number and for the IP Identification fields, and the
attacker knew the data being transferred over the target TCP
connection, he could be able to carefully forge the IP payload of his
IP fragments so that the checksum of the reassembled TCP segment
matched the Checksum included in the TCP header of the first (and
legitimate) IP fragment.

As discussed in Section 4.1 of [CPNI, 2008], IP fragmentation
provides a vector for performing a variety of attacks against an IP
implementation.  Therefore, we discourage the reliance on IP
fragmentation by end-systems, and recommend the implementation of
mechanisms for the discovery of the Path-MTU, such as that described
in Section 15.7.3 of this document and/or that described in RFC 4821
[Mathis and Heffner, 2007].  We nevertheless recommend randomization
of the IP Identification field as described in Section 3.5.2 of
[CPNI, 2008].  While randomization of the IP Identification field
does not eliminate this attack vector, it does require more work on
the side of the attacker to successfully exploit it.

## 16.3.  Broadcast and multicast IP addresses

TCP connection state is maintained between only two endpoints at a
time.  As a result, broadcast and multicast IP addresses should not
be allowed for the establishment of TCP connections.  Section 4.3 of
[CPNI, 2008] provides advice about which specific IP address blocks
should not be allowed for connection-oriented protocols such as TCP.


## 17.  Security Considerations

This document provides a thorough security assessment of the
Transmission Control Protocol (TCP), identifies a number of
vulnerabilities, and specifies possible counter-measures.
Additionally, it provides implementation guidance such that the
resilience of TCP implementations is improved.


## 18.  Acknowledgements

The author would like to thank (in alphabetical order) David Borman,
Wesley Eddy, Alfred Hoenes, and Michael Scharf, for providing
valuable feedback on earlier versions of thi document.

This document is heavily based on the document "Security Assessment
of the Transmission Control Protocol (TCP)" [CPNI, 2009] written by
Fernando Gont on behalf of CPNI (Centre for the Protection of
National Infrastructure).

The author would like to thank (in alphabetical order) Randall
Atkinson, Guillermo Gont, Alfred Hoenes, Jamshid Mahdavi, Stanislav
Shalunov, Michael Welzl, Dan Wing, Andrew Yourtchenko, Michal
Zalewski, and Christos Zoulas, for providing valuable feedback on
earlier versions of the UK CPNI document.

Additionally, the author would like to thank (in alphabetical order)

Mark Allman, David Black, Ethan Blanton, David Borman, James Chacon, John Heffner, Jerrold Leichter, Jamshid Mahdavi, Keith Scott, Bill Squier, and David White, who generously answered a number of questions that araised while the aforementioned document was being written.

Finally, the author would like to thank CPNI (formely NISCC) for their continued support.


## 19. References (to be translated to xml)

Abley, J., Savola, P., Neville-Neil, G. 2007.  Deprecation of Type 0 Routing Headers in IPv6.  RFC 5095.

Allman, M. 2003.  TCP Congestion Control with Appropriate Byte Counting (ABC).  RFC 3465.

Allman, M. 2008.  Comments On Selecting Ephemeral Ports.  Available at: http://www.icir.org/mallman/share/ports-dec08.pdf

Allman, M., Paxson, V., Stevens, W. 1999.  TCP Congestion Control. RFC 2581.

Allman, M., Balakrishnan, H., Floyd, S. 2001.  Enhancing TCP's Loss Recovery Using Limited Transmit.  RFC 3042.

Allman, M., Floyd, S., and C. Partridge. 2002.  Increasing TCP's Initial Window.  RFC 3390.

Baker, F. 1995.  Requirements for IP Version 4 Routers.  RFC 1812.

Baker, F., Savola, P. 2004.  Ingress Filtering for Multihomed Networks.  RFC 3704.

Barisani, A. 2006.  FTester - Firewall and IDS testing tool. Available at: http://dev.inversepath.com/trac/ftester

Beck, R. 2001.  Passive-Aggressive Resistance: OS Fingerprint Evasion.  Linux Journal.

Bellovin, S. M. 1989.  Security Problems in the TCP/IP Protocol Suite.  Computer Communication Review, Vol. 19, No. 2, pp. 32-48.

Bellovin, S. M. 1996.  Defending Against Sequence Number Attacks. RFC 1948.

Bellovin, S. M. 2006.  Towards a TCP Security Option.  IETF Internet-

Draft (draft-bellovin-tcpsec-00.txt), work in progress.

Bernstein, D. J. 1996.  SYN cookies.  Available at:
http://cr.yp.to/syncookies.html

Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., and Weiss,
W., 1998.  An Architecture for Differentiated Services.  RFC 2475.

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

Borman, D. 1997.  Post to the tcp-impl mailing-list.  Message-Id:
<199706061526.KAA01535@frantic.BSDI.COM>.  Available at:
http://www.kohala.com/start/borman.97jun06.txt

Borman, D., Deering, S., Hinden, R. 1999.  IPv6 Jumbograms.  RFC
2675.

Braden, R. 1989.  Requirements for Internet Hosts -- Communication
Layers.  RFC 1122.

Braden, R. 1992.  Extending TCP for Transactions -- Concepts.  RFC
1379.

Braden, R. 1994.  T/TCP -- TCP Extensions for Transactions Functional
Specification.  RFC 1644.

CCSDS. 2006.  Consultative Committee for Space Data Systems (CCSDS)
Recommendation Communications Protocol Specification (SCPS) --
Transport Protocol (SCPS-TP).  Blue Book.  Issue 2.  Available at:
http://public.ccsds.org/publications/archive/714x0b2.pdf

CERT. 1996.  CERT Advisory CA-1996-21: TCP SYN Flooding and IP
Spoofing Attacks.  Available at:
http://www.cert.org/advisories/CA-1996-21.html

CERT. 1997.  CERT Advisory CA-1997-28 IP Denial-of-Service Attacks.
Available at: http://www.cert.org/advisories/CA-1997-28.html

CERT. 2000.  CERT Advisory CA-2000-21: Denial-of-Service
Vulnerabilities in TCP/IP Stacks.  Available at:
http://www.cert.org/advisories/CA-2000-21.html

CERT. 2001.  CERT Advisory CA-2001-09: Statistical Weaknesses in
TCP/IP Initial Sequence Numbers.  Available at:
http://www.cert.org/advisories/CA-2001-09.html

CERT. 2003.  CERT Advisory CA-2003-13 Multiple Vulnerabilities in
Snort Preprocessors.  Available at:
http://www.cert.org/advisories/CA-2003-13.html

Cisco. 2008a.  Cisco Security Appliance Command Reference, Version
7.0.  Available at: http://www.cisco.com/en/US/docs/security/asa/
asa70/command/reference/tz.html#wp1288756

Cisco. 2008b.  Cisco Security Appliance System Log Messages, Version
8.0.  Available at: http://www.cisco.com/en/US/docs/security/asa/
asa80/system/message/logmsgs.html#wp4773952

Clark, D.D. 1982.  Fault isolation and recovery.  RFC 816.

Clark, D.D. 1988.  The Design Philosophy of the DARPA Internet
Protocols, Computer Communication Review, Vol. 18, No.4, pp. 106-114.

Connolly, T., Amer, P., Conrad, P. 1994.  An Extension to TCP :
Partial Order Service.  RFC 1693.

Conta, A., Deering, S., Gupta, M. 2006.  Internet Control Message
Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6)
Specification.  RFC 4443.

CORE. 2003.  Core Secure Technologies Advisory CORE-2003-0307: Snort
TCP Stream Reassembly Integer Overflow Vulnerability.  Available at:
http://www.coresecurity.com/common/showdoc.php?idx=313&idxseccion=10

CPNI, 2008.  Security Assessment of the Internet Protocol.  Available
at: http://www.cpni.gov.uk/Docs/InternetProtocol.pdf

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

daemon9, route, and infinity. 1996.  IP-spoofing Demystified (Trust-
Relationship Exploitation), Phrack Magazine, Volume Seven, Issue
Forty-Eight, File 14 of 18.  Available at:
http://www.phrack.org/archives/48/P48-14

Deering, S., Hinden, R. 1998.  Internet Protocol, Version 6 (IPv6)
Specification.  RFC 2460.

Dharmapurikar, S., Paxson, V. 2005.  Robust TCP Stream Reassembly In
the Presence of Adversaries.  Proceedings of the USENIX Security
Symposium 2005.

Duke, M., Braden, R., Eddy, W., Blanton, E. 2006.  A Roadmap for

Transmission Control Protocol (TCP) Specification Documents.  [RFC 4614].

Ed3f. 2002.  Firewall spotting and networks analisys with a broken CRC.  Phrack Magazine, Volume 0x0b, Issue 0x3c, Phile #0x0c of 0x10. Available at: http://www.phrack.org/phrack/60/p60-0x0c.txt

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

Fenner, B. 2006.  Experimental Values in IPv4, IPv6, ICMPv4, ICMPv6, UDP, and TCP Headers.  [RFC 4727].

Ferguson, P., and Senie, D. 2000.  Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing.  [RFC 2827].

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. 1999.  Hypertext Transfer Protocol -- HTTP/1.1.  [RFC 2616].

Floyd, S., Mahdavi, J., Mathis, M., Podolsky, M. 2000.  An Extension to the Selective Acknowledgement (SACK) Option for TCP.  [RFC 2883].

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

Floyd, S., Allman, M., Jain, A., Sarolahti, P. 2007.  Quick-Start for TCP and IP.  [RFC 4782].

Fyodor. 1998.  Remote OS Detection via TCP/IP Stack Fingerprinting. Phrack Magazine, Volume 8, Issue, 54.

Fyodor. 2006a.  Remote OS Detection via TCP/IP Fingerprinting (2nd Generation).  Available at: http://insecure.org/nmap/osdetect/.

Fyodor. 2006b.  Nmap - Free Security Scanner For Network Exploration and Audit.  Available at: http://www.insecure.org/nmap.

Fyodor. 2008.  Nmap Reference Guide: Port Scanning Techniques. Available at: http://nmap.org/book/man-port-scanning-techniques.html

GIAC. 2000.  Egress Filtering v 0.2.  Available at: http://www.sans.org/y2k/egress.htm

Giffin, J., Greenstadt, R., Litwack, P., Tibbetts, R. 2002.  Covert Messaging through TCP Timestamps.  PET2002 (Workshop on Privacy Enhancing Technologies), San Francisco, CA, USA, April2002.

Available at:
http://web.mit.edu/greenie/Public/CovertMessaginginTCP.ps

Gill, V., Heasley, J., Meyer, D., Savola, P, Pignataro, C. 2007.  The
Generalized TTL Security Mechanism (GTSM).  RFC 5082.

Gont, F. 2006.  Advanced ICMP packet filtering.  Available at:
http://www.gont.com.ar/papers/icmp-filtering.html

Gont, F. 2008a.  ICMP attacks against TCP.  IETF Internet-Draft
(draft-ietf-tcpm-icmp-attacks-04.txt), work in progress.

Gont, F.. 2008b.  TCP's Reaction to Soft Errors.  IETF Internet-Draft
(draft-ietf-tcpm-tcp-soft-errors-09.txt), work in progress.

Gont, F. 2009.  On the generation of TCP timestamps.  IETF Internet-
Draft (draft-gont-tcpm-tcp-timestamps-01.txt), work in progress.

Gont, F., Srisuresh, P. 2008.  Security Implications of Network
Address Translators (NATs).  IETF Internet-Draft
(draft-gont-behave-nat-security-01.txt), work in progress.

Gont, F., Yourtchenko, A. 2009.  On the implementation of TCP urgent
data.  IETF Internet-Draft (draft-gont-tcpm-urgent-data-01.txt), work
in progress.

Heffernan, A. 1998.  Protection of BGP Sessions via the TCP MD5
Signature Option.  RFC 2385.

Heffner, J. 2002.  High Bandwidth TCP Queuing.  Senior Thesis.

Hnes, A. 2007.  TCP options - tcp-parameters IANA registry.  Post to
the tcpm wg mailing-list.  Available at:
http://www.ietf.org/mail-archive/web/tcpm/current/msg03199.html

IANA. 2007.  Transmission Control Protocol (TCP) Option Numbers.
Avialable at: http://www.iana.org/assignments/tcp-parameters/

IANA. 2008.  Port Numbers.  Available at:
http://www.iana.org/assignments/port-numbers

Jacobson, V. 1988.  Congestion Avoidance and Control.  Computer
Communication Review, vol. 18, no. 4, pp. 314-329.  Available at:
ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z

Jacobson, V., Braden, R. 1988.  TCP Extensions for Long-Delay Paths.
RFC 1072.

   Jacobson, V., Braden, R., Borman, D. 1992.  TCP Extensions for High
   Performance.  RFC 1323.

   Jones, S. 2003.  Port 0 OS Fingerprinting.  Available at:
   http://www.gont.com.ar/docs/port-0-os-fingerprinting.txt

   Kent, S. and Seo, K. 2005.  Security Architecture for the Internet
   Protocol.  RFC 4301.

   Klensin, J. 2008.  Simple Mail Transfer Protocol.  RFC 5321.

   Ko, Y., Ko, S., and Ko, M. 2001.  NIDS Evasion Method named SeolMa.
   Phrack Magazine, Volume 0x0b, Issue 0x39, phile #0x03 of 0x12.
   Available at: http://www.phrack.org/issues.html?issue=57&id=3#article

   Lahey, K. 2000.  TCP Problems with Path MTU Discovery.  RFC 2923.

   Lemon, 2002.  Resisting SYN flood DoS attacks with a SYN cache.
   Proceedings of the BSDCon 2002 Conference, pp 89-98.

   Maimon, U. 1996.  Port Scanning without the SYN flag.  Phrack
   Magazine, Volume Seven, Issue Fourty-Nine, phile #0x0f of 0x10.
   Available at:
   http://www.phrack.org/issues.html?issue=49&id=15#article

   Mathis, M., Mahdavi, J., Floyd, S. Romanow, A. 1996.  TCP Selective
   Acknowledgment Options.  RFC 2018.

   Mathis, M., and Heffner, J. 2007.  Packetization Layer Path MTU
   Discovery.  RFC 4821.

   McCann, J., Deering, S., Mogul, J. 1996.  Path MTU Discovery for IP
   version 6.  RFC 1981.

   McKusick, M., Bostic, K., Karels, M., and J. Quarterman. 1996.  The
   Design and Implementation of the 4.4BSD Operating System.  Addison-
   Wesley.

   Meltman. 1997. new TCP/IP bug in win95.  Post to the bugtraq mailing-
   list.  Available at: http://insecure.org/sploits/land.ip.DOS.html

   Miller, T. 2006.  Passive OS Fingerprinting: Details and Techniques.
   Available at: http://www.ouah.org/incosfingerp.htm .

   Mogul, J., and Deering, S. 1990.  Path MTU Discovery.  RFC 1191.

   Morris, R. 1985.  A Weakness in the 4.2BSD Unix TCP/IP Software.
   Technical Report CSTR-117, AT&T Bell Laboratories.  Available at:

   http://pdos.csail.mit.edu/~rtm/papers/117.pdf .

   Myst. 1997.  Windows 95/NT DoS.  Post to the bugtraq mailing-list.
   Available at: http://seclists.org/bugtraq/1997/May/0039.html

   Nichols, K., Blake, S., Baker, F., and Black, D. 1998.  Definition of
   the Differentiated Services Field (DS Field) in the IPv4 and IPv6
   Headers.  RFC 2474.

   NISCC. 2004.  NISCC Vulnerability Advisory 236929: Vulnerability
   Issues in TCP.  Available at:
   http://www.uniras.gov.uk/niscc/docs/re-20040420-00391.pdf

   NISCC. 2005.  NISCC Vulnerability Advisory 532967/NISCC/ICMP:
   Vulnerability Issues in ICMP packets with TCP payloads.  Available
   at: http://www.niscc.gov.uk/niscc/docs/re-20050412-00303.pdf

   NISCC. 2006.  NISCC Technical Note 01/2006: Egress and Ingress
   Filtering.  Available at:
   http://www.niscc.gov.uk/niscc/docs/re-20060420-00294.pdf?lang=en

   Ostermann, S. 2008. tcptrace tool.  Tool and documentation available
   at: http://www.tcptrace.org.

   Paxson, V., Allman, M. 2000.  Computing TCP's Retransmission Timer.
   RFC 2988.

   PCNWG. 2009.  Congestion and Pre-Congestion Notification (pcn)
   charter.  Available at:
   http://www.ietf.org/html.charters/pcn-charter.html

   PMTUDWG. 2007.  Path MTU Discovery (pmtud) charter.  Available at:
   http://www.ietf.org/html.charters/OLD/pmtud-charter.html

   Postel, J. 1981a.  Internet Protocol.  DARPA Internet Program.
   Protocol Specification.  RFC 791.

   Postel, J. 1981b.  Internet Control Message Protocol.  RFC 792.

   Postel, J. 1981c.  Transmission Control Protocol.  DARPA Internet
   Program.  Protocol Specification.  RFC 793.

   Postel, J. 1987.  TCP AND IP BAKE OFF.  RFC 1025.

   Ptacek, T. H., and Newsham, T. N. 1998.  Insertion, Evasion and
   Denial of Service: Eluding Network Intrusion Detection.  Secure
   Networks, Inc. Available at:
   http://www.aciri.org/vern/Ptacek-Newsham-Evasion-98.ps

Ramaiah, A., Stewart, R., and Dalal, M. 2008.  Improving TCP's
Robustness to Blind In-Window Attacks.  IETF Internet-Draft
(draft-ietf-tcpm-tcpsecure-10.txt), work in progress.

Ramakrishnan, K., Floyd, S., and Black, D. 2001.  The Addition of
Explicit Congestion Notification (ECN) to IP.  RFC 3168.

Rekhter, Y., Li, T., Hares, S. 2006.  A Border Gateway Protocol 4
(BGP-4).  RFC 4271.

Rivest, R. 1992.  The MD5 Message-Digest Algorithm.  RFC 1321.

Rowland, C. 1997.  Covert Channels in the TCP/IP Protocol Suite.
First Monday Journal, Volume 2, Number 5.  Available at:
http://www.firstmonday.org/issues/issue2_5/rowland/

Savage, S., Cardwell, N., Wetherall, D., Anderson, T. 1999.  TCP
Congestion Control with a Misbehaving Receiver.  ACM Computer
Communication Review, 29(5), October 1999.

Semke, J., Mahdavi, J., Mathis, M. 1998.  Automatic TCP Buffer
Tuning.  ACM Computer Communication Review, Vol. 28, No. 4.

Shalunov, S. 2000.  Netkill.  Available at:
http://www.internet2.edu/~shalunov/netkill/netkill.html

Shimomura, T. 1995.  Technical details of the attack described by
Markoff in NYT.  Message posted in USENETs comp.security.misc
newsgroup, Message-ID: <3g5gkl$5j1@ariel.sdsc.edu>.  Available at:
http://www.gont.com.ar/docs/post-shimomura-usenet.txt.

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

SinFP. 2006.  Net::SinFP - a Perl module to do OS fingerprinting.
Available at:
http://www.gomor.org/cgi-bin/index.pl?mode=view;page=sinfp

Smart, M., Malan, G., Jahanian, F. 2000.  Defeating TCP/IP Stack
Fingerprinting.  Proceedings of the 9th USENIX Security Symposium,
pp. 229-240.  Available at: http://www.usenix.org/publications/
library/proceedings/sec2000/full_papers/smart/smart_html/index.html

Smith, C., Grundl, P. 2002.  Know Your Enemy: Passive Fingerprinting.
The Honeynet Project.

Spring, N., Wetherall, D., Ely, D. 2003.  Robust Explicit Congestion
Notification (ECN) Signaling with Nonces.  RFC 3540.

Srisuresh, P., Egevang, K. 2001.  Traditional IP Network Address
Translator (Traditional NAT).  RFC 3022.

Stevens, W. R. 1994.  TCP/IP Illustrated, Volume 1: The Protocols.
Addison-Wesley Professional Computing Series.

TBIT. 2001.  TBIT, the TCP Behavior Inference Tool.  Available at:
http://www.icir.org/tbit/

Touch, J. 2007.  Defending TCP Against Spoofing Attacks.  RFC 4953.

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

US-CERT. 2003a.  US-CERT Vulnerability Note VU#26825: Cisco Secure
PIX Firewall TCP Reset Vulnerability.  Available at:
http://www.kb.cert.org/vuls/id/26825

US-CERT. 2003b.  US-CERT Vulnerability Note VU#464113: TCP/IP
implementations handle unusual flag combinations inconsistently.
Available at: http://www.kb.cert.org/vuls/id/464113

US-CERT. 2004a.  US-CERT Vulnerability Note VU#395670: FreeBSD fails
to limit number of TCP segments held in reassembly queue.  Available
at: http://www.kb.cert.org/vuls/id/395670

US-CERT. 2005a.  US-CERT Vulnerability Note VU#102014: Optimistic TCP
acknowledgements can cause denial of service.  Available at:
http://www.kb.cert.org/vuls/id/102014

US-CERT. 2005b.  US-CERT Vulnerability Note VU#396645: Microsoft
Windows vulnerable to DoS via LAND attack.  Available at:
http://www.kb.cert.org/vuls/id/396645

US-CERT. 2005c.  US-CERT Vulnerability Note VU#637934: TCP does not
adequately validate segments before updating timestamp value.
Available at: http://www.kb.cert.org/vuls/id/637934

US-CERT. 2005d.  US-CERT Vulnerability Note VU#853540: Cisco PIX
fails to verify TCP checksum.  Available at:
http://www.kb.cert.org/vuls/id/853540.

Veysset, F., Courtay, O., Heen, O. 2002.  New Tool And Technique For
Remote Operating System Fingerprinting.  Intranode Research Team.

Watson, P. 2004.  Slipping in the Window: TCP Reset Attacks,
CanSecWest 2004 Conference.

Welzl, M. 2008.  Internet congestion control: evolution and current
open issues.  CAIA guest talk, Swinburne University, Melbourne,
Australia.  Available at:
http://www.welzl.at/research/publications/caia-jan08.pdf

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

Zalewski, M. 2001a.  Strange Attractors and TCP/IP Sequence Number
Analysis.  Available at:
http://lcamtuf.coredump.cx/oldtcp/tcpseq.html

Zalewski, M. 2001b.  Delivering Signals for Fun and Profit.
Available at: http://lcamtuf.coredump.cx/signals.txt

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

Zalewski, M. 2003a.  Windows URG mystery solved!  Post to the bugtraq
mailing-list.  Available at:
http://lcamtuf.coredump.cx/p0f-help/p0f/doc/win-memleak.txt

Zalewski, M. 2003b.  A new TCP/IP blind data injection technique?
Post to the bugtraq mailing-list.  Available at:
http://lcamtuf.coredump.cx/ipfrag.txt

Zalewski, M. 2006a. p0f passive fingerprinting tool.  Available at:
http://lcamtuf.coredump.cx/p0f.shtml

Zalewski, M. 2006b. p0f - RST+ signatures.  Available at:
http://lcamtuf.coredump.cx/p0f-help/p0f/p0fr.fp

Zalewski, M. 2007. 0trace - traceroute on established connections.
Post to the bugtraq mailing-list.  Available at:
http://seclists.org/bugtraq/2007/Jan/0176.html

Zalewski, M. 2008.  Museum of broken packets.  Available at:
http://lcamtuf.coredump.cx/mobp/

Zander, S. 2008.  Covert Channels in Computer Networks.  Available
at: http://caia.swin.edu.au/cv/szander/cc/index.html

Zquete, A. 2002.  Improving the functionality of SYN cookies. 6th
IFIP Communications and Multimedia Security Conference (CMS 2002).
Available at: http://www.ieeta.pt/~avz/pubs/CMS02.html

Zweig, J., Partridge, C. 1990.  TCP Alternate Checksum Options.  RFC

1146.

20.  References

20.1.  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.

   [RFC3168]   Ramakrishnan, K., Floyd, S., and D. Black, "The Addition
               of Explicit Congestion Notification (ECN) to IP",
               RFC 3168, September 2001.

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

   [RFC5961]   Ramaiah, A., Stewart, R., and M. Dalal, "Improving TCP's
               Robustness to Blind In-Window Attacks", RFC 5961,
               August 2010.

   [RFC6056]   Larsen, M. and F. Gont, "Recommendations for Transport-
               Protocol Port Randomization", BCP 156, RFC 6056,
               January 2011.

   [RFC6093]   Gont, F. and A. Yourtchenko, "On the Implementation of the
               TCP Urgent Mechanism", RFC 6093, January 2011.

   [RFC6191]   Gont, F., "Reducing the TIME-WAIT State Using TCP
               Timestamps", BCP 159, RFC 6191, April 2011.

   [RFC6528]   Gont, F. and S. Bellovin, "Defending against Sequence
               Number Attacks", RFC 6528, February 2012.

20.2.  Informative References

   [I-D.gont-timestamps-generation]
               Gont, F. and A. Oppermann, "On the generation of TCP
               timestamps", draft-gont-timestamps-generation-00 (work in
               progress), June 2010.

   [I-D.ietf-tcpm-3517bis]
               Blanton, E., Jarvinen, I., Wang, L., Allman, M., Kojo, M.,
               and Y. Nishida, "A Conservative Selective Acknowledgment
               (SACK)-based Loss Recovery Algorithm for TCP",

              draft-ietf-tcpm-3517bis-01 (work in progress),
              January 2012.

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

   [RFC1025]  Postel, J., "TCP and IP bake off", RFC 1025,
              September 1987.

   [RFC1379]  Braden, B., "Extending TCP for Transactions -- Concepts",
              RFC 1379, November 1992.

   [RFC5927]  Gont, F., "ICMP Attacks against TCP", RFC 5927, July 2010.

   [RFC6429]  Bashyam, M., Jethanandani, M., and A. Ramaiah, "TCP Sender
              Clarification for Persist Condition", RFC 6429,
              December 2011.

   [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>, 1995.


## Appendix A.  TODO list

   A Number of formatting issues still have to be fixed in this
   document.  Among others are:

   o  The ASCII-art corresponding to some figures are still missing.  We
      still have to convert the nice JPGs of the UK CPNI document into
      ugly ASCII-art.

   o  The references have not yet been converted to xml, but are
      hardcoded, instead.  That's why they may not look as expected


## Appendix B.  Change log (to be removed by the RFC Editor before publication of this document as an RFC)

### B.1.  Changes from draft-ietf-tcpm-tcp-security-02

   o  Lots of text has been removed out of the document.

   o  The documento track has been changed from BCP to Informational
      (RFC2119-language recommendations ahve been removed).

   o  Where necessary, stand-alone std tracks documents have been
      produced.

B.2.  Changes from draft-ietf-tcpm-tcp-security-01

   A Number of formatting issues still have to be fixed in this
   document.  Among others are:

   o  The whole document was reformatted with RFC 1122 style.


Author's Address

   Fernando Gont
   UK Centre for the Protection of National Infrastructure

   Email: fernando@gont.com.ar
   URI:   http://www.cpni.gov.uk