CPNI
Centre for the Protection of National Infrastructure

CPNI TECHNICAL NOTE 3/2009

SECURITY ASSESSMENT OF THE TRANSMISSION CONTROL PROTOCOL (TCP)

FEBRUARY 2009

Contents

1. Preface 5
   1.1. Introduction 5
   1.2. Scope of this document 6
   1.3. Organisation of this document 7
   1.4. Typographical conventions 7
   1.5 Acknowledgements 7
   1.6. Advice and guidance to vendors 8

2. The Transmission Control Protocol 9

3. TCP header fields 10
   3.1. Source Port 10
   3.2. Destination port 18
   3.3. Sequence number 19
   3.4. Acknowledgement number 20
   3.5. Data Offset 21
   3.6. Control bits 21
   3.7. Window 25
   3.8. Checksum 26
   3.9. Urgent pointer 27
   3.10. Options 31
   3.11. Padding 33
   3.12. Data 33

4. Common TCP options 34
   4.1. End of Option List (Kind = 0) 34
   4.2. No Operation (Kind = 1) 34
   4.3. Maximum Segment Size (Kind = 2) 34
   4.4. Selective Acknowledgement option 36
   4.5. MD5 option (Kind=19) 38
   4.6. Window scale option (Kind = 3) 39
   4.7. Timestamps option (Kind = 8) 40

5. Connection-establishment mechanism 43
   5.1. SYN flood 43
   5.2. Connection forgery 46
   5.3. Connection-flooding attack 47
   5.4. Firewall-bypassing techniques 49

6. Connection-termination mechanism 50
6.1. FIN-WAIT-2 flooding attack 50

7. Buffer management 53
7.1. TCP retransmission buffer 53
7.2. TCP segment reassembly buffer 56
7.3. Automatic buffer tuning mechanisms 58

8. TCP segment reassembly algorithm 62
8.1. Problems that arise from ambiguity in the reassembly process 62

9. TCP congestion control 63
9.1. Congestion control with misbehaving receivers 64
9.2. Blind DupACK triggering attacks against TCP 66
9.3. TCP Explicit Congestion Notification (ECN) 79

10. TCP API 82
10.1 Passive opens and binding sockets 82
10.2. Active opens and binding sockets 83

11. Blind in-window attacks 84
11.1. Blind TCP-based connection-reset attacks 84
11.2. Blind data-injection attacks 90

12. Information leaking 91
12.1. Remote Operating System detection via TCP/IP stack fingerprinting 91
12.2. System uptime detection 94

13. Covert channels 95

14. TCP port scanning 96
14.1. Traditional *connect()* scan 96
14.2. SYN scan 96
14.3. FIN, NULL, and XMAS scans 97
14.4. Maimon scan 98
14.5. Window scan 98
14.6. ACK scan 98

15. Processing of ICMP error messages by TCP 100
15.1. Internet Control Message Protocol 100
15.2. Handling of ICMP error messages 101
15.3 Constraints in the possible solutions 102
15.4. General countermeasures against ICMP attacks 103
15.5. Blind connection-reset attack 104
15.6. Blind throughput-reduction attack 107
15.7. Blind performance-degrading attack 108

16. TCP interaction with the Internet Protocol (IP) 120
16.1. TCP-based traceroute 120
16.2. Blind TCP data injection through fragmented IP traffic 120
16.3. Broadcast and multicast IP addresses 121

17. References 122

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.

There is a clear need for a companion document to the IETF specifications that discusses the security aspects and implications of the protocols, identifies the existing vulnerabilities, discusses the possible countermeasures, and analyses their respective effectiveness.

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 proposed. 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 does not aim to be the final word on the security aspects of TCP. On the contrary, it aims to raise awareness about a number of TCP vulnerabilities that have been faced in the past, those that are currently being faced, and some of those that we may still have to deal with in the future.

Feedback from the community is more than encouraged to help this document be as accurate as possible and to keep it updated as new vulnerabilities are discovered.

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 following IETF RFCs were selected for assessment as part of this work:

- RFC 793, “Transmission Control Protocol. DARPA Internet Program. Protocol Specification” (91 pages)
- RFC 1122, “Requirements for Internet Hosts -- Communication Layers” (116 pages)
- RFC 1191, “Path MTU Discovery” (19 pages)
- RFC 1323, “TCP Extensions for High Performance” (37 pages)
- RFC 1948, “Defending Against Sequence Number Attacks” (6 pages)
- RFC 1981, “Path MTU Discovery for IP version 6” (15 pages)
- RFC 2018, “TCP Selective Acknowledgment Options” (12 pages)
- RFC 2385, “Protection of BGP Sessions via the TCP MD5 Signature Option” (6 pages)
- RFC 2581, “TCP Congestion Control” (14 pages)
- RFC 2675, “IPv6 Jumbograms” (9 pages)
- RFC 2883, “An Extension to the Selective Acknowledgement (SACK) Option for TCP” (17 pages)
- RFC 2884, “Performance Evaluation of Explicit Congestion Notification (ECN) in IP Networks” (18 pages)
- RFC 2988, “Computing TCP’s Retransmission Timer” (8 pages)

- RFC 3168, “The Addition of Explicit Congestion Notification (ECN) to IP” (63 pages)
- RFC 3465, “TCP Congestion Control with Appropriate Byte Counting (ABC)” (10 pages)
- RFC 3517, “A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP” (13 pages)
- RFC 3540, “Robust Explicit Congestion Notification (ECN) Signaling with Nonces” (13 pages)
- RFC 3782, “The NewReno Modification to TCP’s Fast Recovery Algorithm” (19 pages)

1.3. Organisation of this document

This document is basically organised 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.

1.4. Typographical conventions

Throughout this document the header fields of the Transmission Control Protocol (TCP) are discussed in detail. In some cases, a given term may have a slightly different meaning depending on whether it is used to refer to a concept, or to refer to a specific field of the TCP header. In order to avoid any possible confusion arising from this ambiguity, when referring to a specific field of the TCP header the name of the field is written in **`this font type`**.

> Throughout the document there are also a number of parenthetical notes such as this one, to provide additional details or clarifications.

1.5 Acknowledgements

This document was written by Fernando Gont on behalf of CPNI.

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

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

1.6. Advice and guidance to vendors

Vendors are urged to contact CSIRTUK (csirt@cpni.gsi.gov.uk) if they think they may be affected by the issues described in this document. As the lead coordination centre for these issues, CPNI is well placed to give advice and guidance as required.

CPNI works extensively with government departments and agencies, commercial organisations and the academic community to research vulnerabilities and potential threats to IT systems especially where they may have an impact on Critical National Infrastructure's (CNI).

Other ways to contact CPNI, plus CPNI's PGP public key, are available at http://www.cpni.gov.uk.

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. Figure 1 illustrates where TCP fits in the DARPA reference model.

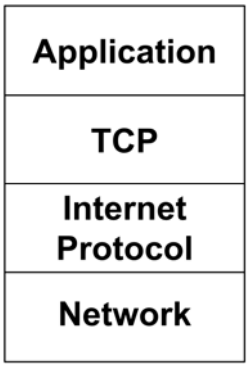

Figure 1: TCP in the DARPA reference model

TCP provides facilities in the following areas:

- Basic Data Transfer
- Reliability
- Flow Control
- Multiplexing
- Connections
- Precedence and Security
- Congestion Control

The core TCP specification, RFC 793 [Postel, 1981c], dates back to 1981 and standardises the basic mechanisms and policies of TCP. RFC 1122 [Braden, 1989] provides clarifications and errata for the original specification. RFC 2581 [Allman et al, 1999] 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 [Postel, 1981c] defines the syntax of a TCP segment, along with the semantics of each of the header fields. Figure 2 illustrates the syntax of a TCP segment.

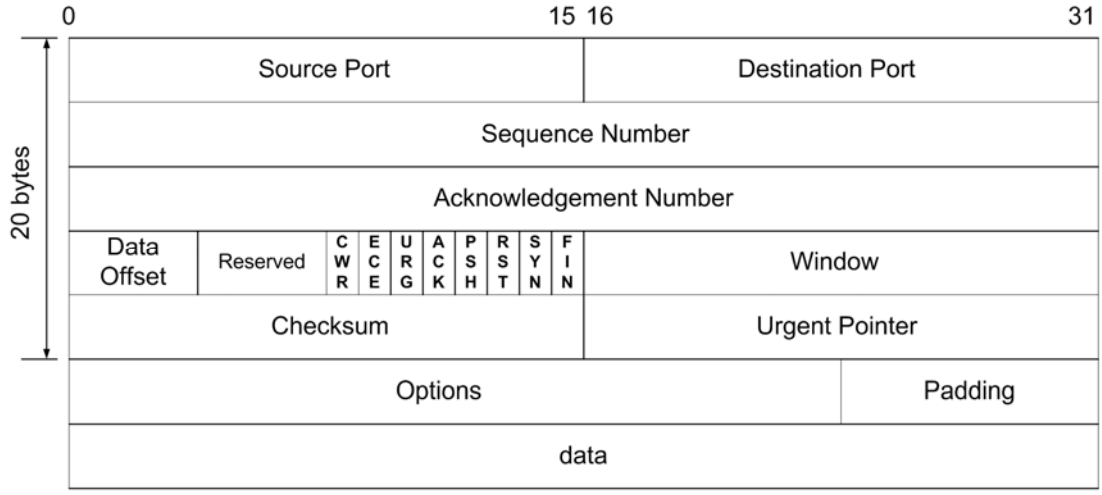

Figure 2: Transmission Control Protocol header format

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

This field contains a 16-bit number that identifies the TCP end-point that originated this TCP segment. Being a 16-bit field, it can contain any value in the range 0-65535.

The Internet Assigned Numbers Authority (IANA) has traditionally reserved the following use of the 16-bit port range of TCP [IANA, 2008]:

- The Well Known Ports, 0 through 1023
- The Registered Ports, 1024 through 49151
- The Dynamic and/or Private Ports, 49152 through 65535

The range of assigned ports managed by the IANA is 0-1023, with the remainder being registered by IANA but not assigned [IANA, 2008]. It is also worth noting that, while some systems restrict use of the port numbers in the range 0-1024 to privileged users, no trust should be granted based on the port numbers used for a TCP connection.

Servers usually bind specific ports on which specific services are usually provided, while clients usually make use of the so-called “ephemeral ports” for the source port of their outgoing connections with the only requirement that the resulting four-tuple must be unique (not currently in use by any other transport protocol instance).

> *While the only requirement for a selected ephemeral port is that the resulting four-tuple (connection-id) is unique, in practice it may be necessary to not allow the allocation of port numbers that are in use by a TCP that is in the LISTEN or CLOSED states for use as ephemeral ports, as this might allow an attacker to “steal” incoming connections from a local server application. Section 10.2 of this document provides a detailed discussion of this issue.*
>
> *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 should not assume that clients use port number from only the Dynamic or Registered port ranges.*

While port 0 is a legitimate port number, it has a special meaning in the UNIX Sockets API. For example, when a TCP port number of 0 is passed as an argument to the *bind()* function, rather than binding port 0, an ephemeral port is selected for the corresponding TCP end-point. As a result, the TCP port number 0 is never actually used in TCP segments.

Different implementations have been found to respond differently to TCP segments that have a port number of 0 as the **`Source Port`** and/or the **`Destination Port`**. As a result, TCP segments with a port number of 0 are usually employed for remote OS detection via TCP/IP stack fingerprinting [Jones, 2003].

Since in practice TCP port 0 is not used by any legitimate application and is only used for fingerprinting purposes, a number of host implementations already reject TCP segments that use 0 as the **`Source Port`** and/or the **`Destination Port`**. Also, a number firewalls filter (by default) any TCP segments that contain a port number of zero for the **`Source Port`** and/or the **`Destination Port`**.

We therefore recommend that TCP implementations respond to incoming TCP segments that have a **`Source Port`** of 0 with an RST (provided these incoming segments do not have the **RST** bit set).

> *Responding with an RST segment to incoming segments that have the **RST** bit would open the door to RST-war attacks.*

As discussed in Section 3.2, we also recommend TCP implementations to respond with an RST to incoming packets that have a **`Destination Port`** of 0 (provided these incoming segments do not have the **RST** bit set).

3.1.1. Problems that may arise as a result of collisions of connection-id's

A number of implementations will not allow the creation of a new connection if there exists a previous incarnation of the same connection in any state other than the fictional state CLOSED. This can be problematic in scenarios in which a client establishes connections with a specific service at a particular server at a high rate: even if the connections are also closed at a high rate, one of the systems (the one performing the active close) will keep each of the closed connections in the TIME-WAIT state for 2**MSL*.

> *MSL (Maximum Segment Lifetime) is the maximum amount of time that a TCP segment can exist in an internet. It is defined to be 2 minutes by RFC 793 [Postel, 1981c].*

If the connection rate is high enough, at some point all the ephemeral ports at the client will be in use by some connection in the TIME-WAIT state, thus preventing the establishment of new connections. In order to overcome this problem, a number of TCP implementations include some heuristics to allow the creation of a new incarnation of a connection that is in the TIME-WAIT state. In such implementations a new incarnation of a previous connection is allowed if:

- The incoming SYN segment contains a timestamp option, and the timestamp is greater than the last timestamp seen in the previous incarnation of the connection (for that direction of the data transfer), or,
- The incoming SYN segment does not contain a timestamp option, but its Initial Sequence Number (ISN) is greater than the last sequence number seen in the previous incarnation of the connection (for that direction of the data transfer).

Unfortunately, these heuristics are optional, and thus cannot be relied upon. Additionally, as indicated by [Silbersack, 2005], if the Timestamp or the ISN are trivially randomised, these heuristics might fail.

> *Section 3.3.1 and Section 4.7.1 of this document recommend algorithms for the generation of TCP Initial Sequence Numbers and TCP timestamps, respectively, that provide randomisation, while still allowing the aforementioned heuristics to work.*

Therefore, the only strategy that can be relied upon to avoid this interoperability problem is to minimise the rate of collisions of connection-id's. A good algorithm to minimise rate of collisions of connection-id's would consider the time a given four-tuple {**`Source Address`**, **`Source Port`**, **`Destination Address`**, **`Destination Port`**} was last used, and would try avoid reusing it for 2**MSL*. However, an efficient implementation approach for this algorithm has not yet been devised. A simple approach to minimise the rate collisions of connection-id's in most scenarios is to maximise the port reuse cycle, such that a port number is not reused before all the other port numbers in the ephemeral port range have been used for outgoing connections. This is the traditional ephemeral port selection algorithm in 4.4BSD implementations.

However, if a single global variable is used to keep track of the last ephemeral port selected, ephemeral port numbers become trivially predictable.

Section 3.1.2 of this document analyses a number of approaches for obfuscating the TCP ephemeral ports, such that the chances of an attacker of guessing the ephemeral ports used for future connections are reduced, while still reducing the probability of collisions of connection-id's. Finally, Section 3.1.3 makes recommendations about the port range that should be used for the ephemeral ports.

3.1.2. Port randomisation algorithms

Since most "blind" attacks against TCP require the attacker to guess or know the four-tuple that identifies the TCP connection to be attacked [Gont, 2008a] [Touch, 2007] [Watson, 2004], obfuscation of this four-tuple to an off-path attacker requires, in a number of scenarios, much more work on the side of the attacker to successfully perform any of these attacks against a TCP connection. Therefore, we recommend that TCP implementations randomise their ephemeral ports.

There are a number of factors to consider when designing a policy of selection of ephemeral ports, which include:

- Minimising the predictability of the ephemeral port numbers used for future connections.
- Minimising the rate of collisions of connection-id's.
- Avoiding conflicts with applications that depend on the use of specific port numbers.

Given the goal of improving TCP's resistance to attack by obfuscation of the four-tuple that identifies a TCP connection, it is key to minimise the predictability of the ephemeral ports that will be selected for new connections. While the obvious approach to address this requirement would be to select the ephemeral ports by simply picking a random value within the chosen ephemeral port number range, this straightforward policy may lead to a short reuse cycle of port numbers, which could lead to the interoperability problems discussed in [Silbersack, 2005].

> *It is also worth noting that, provided adequate randomisation algorithms are in use, the larger the range from which ephemeral pots are selected, the smaller the chances of an attacker are to guess the selected port number. This is discussed in Section 3.1.3 of this document.*

[Larsen and Gont, 2008] provides a detailed discussion of a number of algorithms for obfuscating the ephemeral ports. The properties of these algorithms have been empirically analysed in [Allman, 2008].

[Larsen and Gont, 2008] recently suggested an approach that is meant to comply with the requirements stated above, which resembles the proposal in RFC 1948 [Bellovin, 1996] for selecting TCP Initial Sequence Numbers. Basically, it proposes to give each triple {**Source Address**, **Destination Address**, **Destination Port**} a separate port number space, by selecting ephemeral ports by means of an expression of the form:

port = min_port + (counter + F()) % (max_port - min_port + 1)

Equation 1: Simple hash-based ephemeral port selection algorithm

where:

- port: Ephemeral port number selected for this connection.
- min_port: Lower limit of the ephemeral port number space.
- max_port: Upper limit of the ephemeral port number space.
- counter: A variable that is initialised to some arbitrary value, and is incremented once for each port number that is selected.
- F(): A hash function that should take as input both the local and remote IP addresses, the TCP destination port, and a secret key. The result of F should not be computable without the knowledge of all the parameters of the hash function.

The hash function *F()* separates the port number space for each triple {**Source Address**, **Destination Address**, **Destination Port**} by providing an "offset" in the port number space that is unique (assuming no hash collisions) for each triple. As a result, subsequent connections to the same end-point would be assigned incremental port numbers, thus maximising the port reuse cycle while still making it difficult for an attacker to guess the selected ephemeral port number used for connections with other endpoints.

Keeping track of the last ephemeral port selected for each of the possible values of *F()* would require a considerable amount of system memory. Therefore, a possible approach would be to keep a global *counter* variable, which would reduce the required system memory at the expense of a shorter port reuse cycle. This latter approach would have the same port reuse properties than the widely implemented approach of selecting ephemeral port numbers incrementally (without randomisation), while still reducing the predictability of ephemeral port numbers used for connections with other endpoints. Figure 3 shows this algorithm in pseudo-code.

```
/* Initialization code at system boot time. *
 * Initialization value could be random.    */
counter = 0;

/* Ephemeral port selection function */
                    num_ephememeral = max_port - min_port + 1;
offset = F(local_IP, remote_IP, remote_port, secret_key);
count = num_ephemeral;

do {
   port = min_port + (counter + offset) % num_ephemeral;
   counter++;

   if(four-tuple is unique)
      return port;

   count--;

} while (count > 0);
```

Figure 3: Simple hash-based ephemeral port selection algorithm

An analysis of a sample scenario can help to understand how this algorithm works. Table 2 illustrates, for a number of consecutive connection requests, some possible values for each of the variables used in this ephemeral port selection algorithm. Additionally, the table shows the result of the port selection function.

Nr.	IP address:port	offset	min_port	max_port	counter	port
#1	10.0.0.1:80	1000	1024	65535	1024	3048
#2	10.0.0.1:80	1000	1024	65535	1025	3049
#3	192.168.0.1:80	4500	1024	65535	1026	6550
#4	192.168.0.1:80	4500	1024	65535	1027	6551
#5	10.0.0.1:80	1000	1024	65535	1028	3052

Table 1: Sample scenario for a simple hash-based port randomisation algorithm

The first two entries of the table illustrate the contents of each of the variables when two ephemeral ports are selected to establish two consecutive connections to the same remote end-point {10.0.0.1, 80}. The two ephemeral ports that get selected belong to the same port number “sequence”, since the result of the hash function *F()* is the same in both cases. The second and third entries of the table illustrate the contents of each of the variables when the algorithm later selects two ephemeral ports to establish two consecutive connections to the remote end-point {192.168.0.1, 80}. The result of *F()* is the same for these two cases, and thus the two ephemeral ports that get selected belong to the same “sequence”.

However, this sequence is different from that of the first two port numbers selected before, as the value of *F()* is different from that obtained when those two ports numbers (#1 and #2) were selected earlier. Finally, in entry #5 another ephemeral port is selected to connect to the same end-point as in entries #1 and #2. We note that the selected port number belongs to the same sequence as the first two port numbers selected (#1 and #2), but that two ports of that sequence (3050 and 3051) have been skipped. This is the consequence of having a single global *counter* variable that gets incremented whenever a port number is selected. When *counter* is incremented as a result of the port selections #3 and #4, this causes two ports (3050 and 3051) in all the other the port number sequences to be "skipped", unnecessarily.

[Larsen and Gont, 2008] describes an improvement to this algorithm, in which a value derived from the three-tuple {**`Source Address`**, **`Destination Address`**, **`Destination Port`**} is used as an index into an array of "*counter*" variables, which would be used in the equation described above. The rationale of this approach is that the selection of an ephemeral port number for a given three-tuple {**`Source Address`**, **`Destination Address`**, **`Destination Port`**} should not necessarily cause the *counter* variables corresponding to other three-tuples to be incremented. Figure 4 illustrates this improved algorithm in pseudo-code.

```
/* Initialization at system boot time */
for(i = 0; i < TABLE_LENGTH; i++)
    table[i] = random() % 65536;


/* Ephemeral port selection function */
num_ephemeral = max_port - min_port + 1;
offset = F(local_IP, remote_IP, remote_port, secret_key1);
index = G(local_IP, remote_IP, remote_port, secret_key2);
count = num_ephemeral;

do {
    port = min_port + (offset + table[index]) % num_ephemeral;
    table[index]++;

    if(four-tuple is unique)
    return port;

   count--;

} while (count > 0);
```

Figure 4: Double hash-based ephemeral port selection algorithm

Table 2 illustrates a possible result for the same sequence of events as those in Table 1, along with the values for each of the involved variables.

Nr.	IP address:port	offset	min_port	max_port	index	table[index]	port
#1	10.0.0.1:80	1000	1024	65535	10	1024	3048
#2	10.0.0.1:80	1000	1024	65535	10	1025	3049
#3	192.168.0.1:80	4500	1024	65535	15	1024	6548
#4	192.168.0.1:80	4500	1024	65535	15	1025	6549
#5	10.0.0.1:80	1000	1024	65535	10	1026	3050

Table 2: Sample scenario for a double hash-based port randomisation algorithm

The table illustrates that the destination end-points “10.0.0.1:80” and “192.168.0.1:80” result in different values for *index* and therefore the increments in one of the port number sequence does not affect the other sequences, thus minimising the port reuse frequency.

We recommend the implementation of the ephemeral port selection algorithm illustrated in Figure 4.

3.1.3. TCP ephemeral port range

We recommend that TCP select ephemeral ports from the range 1024-65535 (i.e., set *min_port* and the *max_port* variables of the previous section to 1024 and 65535, respectively). This maximises the port number space from which the ephemeral ports are selected, while intentionally excluding the port numbers in the range 0-1023, which in UNIX systems have traditionally required super-user privileges to bind them.

> *4.4BSD implementations have traditionally chosen ephemeral ports from the range 1024-5000, thus greatly increasing the chances of an attacker of guessing the selected port number [Wright and Stevens, 1994]. Unfortunately, most current implementations are still using a small range of the whole port number space, such as 1024-49151 or 49152-65535.*

It is important to note that a number of applications rely on binding specific port numbers that may be within the ephemeral ports range. If such an application was run while the corresponding port number was in use, the application would fail.

> *This problem does not arise from port randomisation itself, and has actually been experienced by users of popular TCP implementations that do not actually randomise their ephemeral ports.*

A solution to this potential problem would be to maintain a list of port numbers that are usually needed for running popular applications. In case the port number selected by Equation 1 was in such a list, the next available port number would be selected, instead. This “list” of port numbers could be implemented as an array of bits, in which each bit would correspond to each of the 65536 TCP port numbers, with a value of 0 (zero) meaning that the corresponding TCP port is available for allocation as an ephemeral port, and a value of 1 (one) meaning that the corresponding port number should not be allocated as an ephemeral port. The specification of which ports should be “reserved” for applications may depend on the underlying operating system, and is out of the scope of this document.

As discussed in Section 3.1 and Section 10.2, in practice it may be necessary to not allow the allocation as "ephemeral ports" of those port numbers that are currently in use by a TCP that is in the LISTEN or CLOSED states, as this might allow an attacker to “steal” incoming connections from a local server application. Section 10.2 of this document provides a detailed discussion of this issue.

3.2. Destination port

This field contains the destination TCP port of this segment. Being a 16-bit value, it can contain any value in the range 0-65535. While some systems restrict use of the ports numbers in the range 0-1023 to privileged users, no trust should be granted based on the port numbers in use for a connection.

As noted in Section 3.1 of this document, while port 0 is a legitimate port number, it has a special meaning in the UNIX Sockets API. For example, when a TCP port number of 0 is passed as an argument to the *bind()* function, rather than binding port 0, an ephemeral port is selected for the corresponding TCP end-point. As a result, the TCP port number 0 is never actually used in TCP segments.

Different implementations have been found to respond differently to TCP segments that have a port number of 0 as the **`Source Port`** and/or the **`Destination Port`**. As a result, TCP segments with a port number of 0 are usually employed for remote OS detection via TCP/IP stack fingerprinting [Jones, 2003].

Since in practice TCP port 0 is not used by any legitimate application and is only used for fingerprinting purposes, a number of host implementations already reject TCP segments that use 0 as the **`Source Port`** and/or the **`Destination Port`**. Also, a number firewalls filter (by default) any TCP segments that contain a port number of zero for the **`Source Port`** and/or the **`Destination Port`**.

We therefore recommend that TCP implementations respond to incoming TCP segments that have a **`Destination Port`** of 0 with an RST (provided these incoming segments do not have the **`RST`** bit set).

> *Responding with an RST segment to incoming packets that have the* **`RST`** *bit would open the door to RST-war attacks.*

Some systems have been found to be unable to process TCP segments in which the source endpoint {**`Source Address`**, **`Source Port`**} is the same than the destination end-point {**`Destination Address`**, **`Destination Port`**}. Such TCP segments have been reported to cause malfunction of a number of implementations [CERT, 1996], and have been exploited in the past to perform Denial of Service (DoS) attacks [Meltman, 1997]. While these packets are extremely unlikely to exist in real and legitimate scenarios, TCP should nevertheless be able to process them without the need of any “extra” code.

*A SYN segment in which the source end-point {**`Source Address`**, **`Source Port`**} is the same as the destination end-point {**`Destination Address`**, **`Destination Port`**} will result in a "simultaneous open" scenario, such as the one described in page 32 of RFC 793 [Postel, 1981c]. Therefore, those TCP implementations that correctly handle simultaneous opens should already be prepared to handle these unusual TCP segments.*

3.3. Sequence number

This field contains the sequence number of the first data octet in this segment. If the SYN flag is set, the sequence number is the Initial Sequence Number (ISN) of the connection, and the first data octet has the sequence number ISN+1.

3.3.1. Generation of Initial Sequence Numbers

The choice of the Initial Sequence Number of a connection is not arbitrary, but aims to minimise the chances of a stale segment from being accepted by a new incarnation of a previous connection. RFC 793 [Postel, 1981c] suggests the use of a global 32-bit ISN generator, whose lower bit is incremented roughly every 4 microseconds.

However, use of such an ISN generator makes it trivial to predict the ISN that a TCP will use for new connections, thus allowing 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 [Morris, 1985], and its exploitation was widely publicised about 10 years later [Shimomura, 1995].

As a matter of fact, protection against old stale segments from a previous incarnation of the connection comes from allowing the creation of a new incarnation of a previous connection only after 2**MSL* have passed since a segment corresponding to the old incarnation was last seen. This is accomplished by the TIME-WAIT state, and TCP's "quiet time" concept. However, as discussed in Section 3.1 and Section 11.1.2 of this document, the ISN can be used to perform some heuristics meant to avoid an interoperability problem that may arise when two systems establish connections at a high rate. In order for such heuristics to work, the ISNs generated by a TCP should be monotonically increasing.

RFC 1948 [Bellovin, 1996] proposed a scheme that greatly reduces the chances of an attacker from guessing the ISN of a TCP, while still producing a monotonically-increasing sequence that allows implementation of the optimisation described in Section 3.1 and Section 11.1.2 of this document. Basically, the document proposes to compute the ISN of a new connection as a result of the expression:

ISN = M + F(localhost, localport, remotehost, remoteport, secret_key)

where *M* is a monotonically increasing counter maintained within TCP, and *F()* is a hash function. As it is vital that *F()* not be computable from the outside, RFC 1948 [Bellovin, 1996] suggests it to be a cryptographic hash function of the connection-id and some secret data.

> *RFC 1948 [Bellovin, 1996] proposes that F() be a MD5 hash function applied to the connection-id and some secret data. While there have been concerns regarding the properties of MD5 as a hash function, in this case it is simply used for obfuscating the ISN, rather than for signing the data contained in the TCP segments. While the MD5 function could be replaced by a more secure hash function, at the point in which this issue becomes a concern, proper authentication mechanisms such as IPsec [Kent and Seo, 2005] should be considered for protecting the corresponding TCP connection.*

[CERT, 2001] and [US-CERT, 2001] are advisories about the security implications of weak ISN generators. [Zalewski, 2001a] and [Zalewski, 2002] contain a detailed analysis of ISN generators, and a survey of the algorithms in use by popular TCP implementations.

Finally, 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". This information leakage could be eliminated by rewriting the contents of all those header fields and options that make use of sequence numbers (such as the **Sequence Number** and the **Acknowledgement Number** fields, and the SACK Option) at the NAT. [Gont and Srisuresh, 2008] provides a detailed discussion of the security implications of NATs and of the possible mitigations for this and other issues.

3.4. 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. That is, the following expression must be true:

SEG.ACK <= SND.NXT

As a result of recent concerns on forgery attacks against TCP (see Section 11 of this document), ongoing work at the IETF [Ramaiah et al, 2008] has proposed to enforce a more strict check on the **Acknowledgement Number**. The following check should be enforced on segments that have the ACK bit set:

SND.UNA - SND.MAX.WND <= SEG.ACK <= SND.NXT

If a TCP segment does not pass this check, the segment should be dropped, and an ACK segment should be sent in response.

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.

3.5. Data Offset

The **`Data Offset`** field indicates the length of the TCP header in 32-bit words. As the minimum TCP header size is 20 bytes, the minimum legal value for this field is 5. Therefore, the following check should be enforced:

`Data Offset` >= 5

For obvious reasons, the TCP header cannot be larger than the whole TCP segment it is part of. Therefore, the following check should be enforced:

`Data Offset` * 4 <= TCP segment length

> *The TCP segment length should be obtained from the IP layer, as TCP does not include a TCP segment length field.*

3.6. 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 [Postel, 1987] [Braden, 1992]. 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.

3.6.1. 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.

Therefore, we recommend implementations to simply ignore the **`Reserved`** field.

3.6.2. 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 synchronised 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.

3.6.3. 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.

Once a TCP connection has been established, an ACK segment with the ECE bit set indicates that congestion was encountered in the network on the path from the sender to the receiver. This indication of congestion should be treated just as a congestion loss in non-ECN-capable TCP [Ramakrishnan et al, 2001]. Additionally, TCP should not increase the congestion window (*cwnd*) in response to such an ACK segment that indicates congestion, and should also not react to congestion indications more than once every window of data (or once per round-trip time).

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

3.6.4. 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.

3.6.5. 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.

3.6.6. PSH

RFC 793 [Postel, 1981c] contains (in pages 54-64) a functional description of a TCP Application Programming Interface (API). One of the parameters of the SEND function is the PUSH flag which, when set, signals the local TCP that it must send all unsent data. The TCP **PSH** (PUSH) flag will be set in the last outgoing segment, to signal the push function to the receiving TCP. Upon receipt of a segment with the PSH flag set, the receiving user's buffer is returned to the user, without waiting for additional data to arrive.

There are two security considerations arising from the PUSH function. On the sending side, an attacker could cause a large amount of data to be queued for transmission without setting the PUSH flag in the SEND call. This would prevent the local TCP from sending the queued data, causing system memory to be tied to those data for an unnecessarily long period of time.

An analogous consideration should be made for the receiving TCP. TCP is allowed to buffer incoming data until the receiving user's buffer fills or a segment with the **PSH** bit set is received. If the receiving TCP implements this policy, an attacker could send a large amount of data, slightly less than the receiving user's buffer size, to cause system memory to be tied to these data for an unnecessarily long period of time. Both of these issues are discussed in Section 4.2.2.2 of RFC 1122 [Braden, 1989].

In order to mitigate these potential vulnerabilities, we suggest assuming an implicit "PUSH" in every SEND call. On the sending side, this means that as a result of a SEND call TCP should try to send all queued data (provided that TCP's flow control and congestion control algorithms allow it). On the receiving side, this means that the received data will be immediately delivered to an application calling the RECEIVE function, even if the data already available are less than those requested by the application.

It is interesting to note that popular TCP APIs (such as "sockets") do not provide a PUSH flag in any of the interfaces they define, but rather perform some kind of "heuristics" to set the **PSH** bit in outgoing segments. As a result, the value of the **PSH** bit in the received TCP segments is usually a policy of the sending TCP, rather than a policy of the sending application. All robust applications that make use of those APIs (such as the sockets API) properly handle the case of a RECEIVE call returning less data (e.g., zero) than requested, usually by performing subsequent RECEIVE calls.

Another potential malicious use of the **PSH** bit would be for an attacker to send small TCP segments (probably with zero bytes of data payload) to cause the receiving application to be unnecessarily woken up (increasing the CPU load), or to cause malfunction of poorly-written applications that may not handle well the case of RECEIVE calls returning less data than requested.

3.6.7. RST

The RST bit is used to request the abortion (abnormal close) of a TCP connection. RFC 793 [Postel, 1981c] 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], [Ramaiah et al, 2008] suggests the following alternative processing rules for RST segments:

- If the **Sequence Number** of the RST segment is not valid (i.e., falls outside of the receive window), silently drop the segment.
- If the **Sequence Number** of the RST segment matches the next expected sequence number (RCV.NXT), abort the corresponding connection.
- If the **Sequence Number** is valid (i.e., falls within the receive window) but is not exactly RCV.NXT, send an ACK segment (a “challenge ACK”) of the form: <SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>

[Ramaiah et al, 2008] suggests that implementations should rate-limit the challenge ACK segments sent as a result of implementation of this mechanism.

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

3.6.8. SYN

The SYN bit is used during the connection-establishment phase, to request the synchronisation 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.

3.6.9. 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.

3.7. 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.*

3.7.1. Security implications of the maximum TCP window size

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.

3.7.2. Security implications arising from closed windows

The TCP window is a flow-control mechanism that prevents a fast data sender application from overwhelming a "slow" receiver. 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.

3.8. Checksum

The **`Checksum`** field is an error detection mechanism meant for the contents of the TCP segment and a number of important fields of the IP header. It is computed over the full TCP header pre-pended with a pseudo header that includes the IP **`Source Address`**, the IP **`Destination Address`**, the **`Protocol`** number, and the TCP segment length. While in principle there should not be security implications arising from this 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.

3.9. Urgent pointer

If the **Urgent** bit is set, the **Urgent Pointer** field communicates the current value of the urgent pointer as a positive offset from the **Sequence Number** in this segment. That is, the urgent pointer is obtained as:

urgent_pointer = **Sequence Number** + **Urgent Pointer**

According to RFC 1122 [Braden, 1989], the urgent pointer (urgent_pointer) points to the last byte of urgent data in the stream. However, in virtually all TCP implementations the urgent pointer has the semantics of pointing to the byte following the last byte of urgent data [Gont and Yourtchenko, 2009].

> *There was some ambiguity in RFC 793 [Postel, 1981c] with respect to the semantics of the urgent pointer. Section 4.2.2.4 of RFC 1122 [Braden, 1989] clarified this ambiguity, stating that the urgent pointer points to the last byte of urgent data. However, the RFC 1122 semantics for the urgent pointer never resulted into actual implementations.*
>
> *Ongoing work at the IETF [Gont and Yourtchenko, 2009] aims at updating the IETF specifications to change the semantics of the urgent pointer so that it points to "the byte following the last byte of urgent data", thus accommodating virtually all existing implementations of the TCP urgent mechanism.*

Section 3.7 of RFC 793 [Postel, 1981c] states (in page 42) that to send an urgent indication the user must also send at least one byte of data. Therefore, if the **URG** bit is set, the following check should be performed:

Segment.Size - **Data Offset** * 4 > 0

If a TCP segment with the **URG** bit set does not pass this check, it should be silently dropped.

It is worth noting that the resulting urgent_pointer may refer to a sequence number not present in this segment. That is, the “last byte of urgent data” might be received in successive segments.

If the **URG** bit is zero, the **Urgent Pointer** is not valid, and thus should not be processed by the receiving TCP. Nevertheless, we recommend TCP implementations to set the **Urgent Pointer** to zero when sending a TCP segment that does not have the **URG** bit set, and to ignore the **Urgent Pointer** (as required by RFC 793) when the **URG** bit is zero.

> *Some stacks have been known to fail to set the* **Urgent Pointer** *to zero when the* **URG** *bit is zero, thus leaking out the corresponding system memory contents. [Zalewski, 2008] provides further details about this issue.*

According to the IETF specifications, TCP’s urgent mechanism simply marks an interesting point in the data stream that applications may want to skip to even before processing any other data. However, “urgent data” must still be delivered “in band” to the application.

Unfortunately, virtually all TCP implementations process TCP urgent data differently. By default, the “last byte of urgent data” is delivered to the application “out of band”. That is, it is not delivered as part of the normal data stream.

> *For example, the “out of band” byte is read by an application when a recv(2) system call with the MSG_OOB flag set is issued.*

Most implementations provide a socket option (*SO_OOBINLINE*) that allows an application to override the default processing of urgent data, so that they are delivered “in band” to the application, thus providing the semantics intended by the IETF specifications.

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.

The following subsections analyze the security implications of the TCP urgent mechanism. Section 3.9.1 discusses the security implications arising from the different possible semantics for the urgent pointer and for the TCP urgent indications. Section 3.9.2 discusses the security implications that may arise when systems implement the TCP urgent mechanism as “out of band” data.

3.9.1. Security implications arising from ambiguities in the processing of urgent indications

As discussed in Section 3.9, there exists some ambiguity with respect to how a receiving application may process the TCP urgent indications sent by the peer application. Firstly, the different possible semantics of the urgent pointer create ambiguity with respect to which of the bytes in the data stream are considered to be “urgent data”. Secondly, some applications may process these urgent data “in band” (either if TCP urgent data is implemented as intended by the IETF specifications, or if the application sets the *SO_OOBINLINE* socket option), while others may process them “out of band” (e.g., as a result of a *recv(2)* call with the *MSG_OOB* option set). Thirdly, some TCP implementations keep a buffer of a single byte for storing the “urgent byte” that is delivered “out of band” to the application. Thus, if successive indications of urgent data are received before the application reads the pending “out of band” byte, the pending byte will be discarded (i.e., overwritten by the new byte of urgent data). Fourthly, some middle-boxes clear the **`URG`** bit and reset the **`Urgent`** field to zero before forwarding a packet, thus essentially eliminating the “urgent” indication.

> [Cisco, 2008a] provides documentation of such a middle-box.

All these considerations make it difficult for Network Intrusion Detection Systems (NIDS) to monitor the application-layer data stream transferred to the screened systems, thus potentially leading to false negatives or false positives.

> *[Ko et al, 2001] describes some of the possible ways to exploit TCP urgent data to evade Network Intrusion Detection Systems (NIDS).*

Considering the security implications of the TCP urgent mechanism, and given that widely-deployed middle-boxes clear the **`URG`** bit and reset the **`Urgent Pointer`** to zero (thus making the urgent indication unreliable), we discourage the use of the TCP urgent mechanism by applications.

We also recommend that those legacy applications that depend on the TCP urgent mechanism set the *SO_OOBINLINE* socket option, so that urgent data are delivered “in band” to the application running on top of TCP.

Packet scrubbers might consider clearing the **`URG`** bit, and setting the **`Urgent Pointer`** to zero, thus eliminating the urgent indication and causing urgent data to be processed in-line regardless of the semantics in use at the destination system for the TCP urgent indications. However, this might cause interoperability problems and/or undesired behavior that should be considered before enabling such behavior in packet scrubbers.

3.9.2 Security implications arising from the implementation of the urgent mechanism as “out of band” data

As described in the previous sub-section, some implementations keep a buffer of a single byte for storing the “urgent byte” that is delivered “out of band” to the application running on top of TCP. If successive indications of urgent data are received before the application reads the pending “urgent” byte, the pending byte is discarded (i.e., overwritten by the new byte of urgent data). This makes it difficult for a NIDS to track the application-layer data transferred to

the monitored system, as some of the urgent data might (or might not) end up being discarded at the destination system, depending on the timing of the arriving segments and the consumption of urgent data by the application (assuming the *SO_OOBINLINE* socket option has not been set).

In order to avoid urgent data being discarded, some implementations queue each of the received “urgent bytes”, so that even if another urgent indication is received before the pending urgent data are consumed by the application, those bytes do not need to be discarded. Unfortunately, some of these implementations have been known to fail to enforce any limits on the amount of urgent data that they queue. As a result, an attacker could exhaust the kernel memory of such TCP implementations by sending successive TCP segments that carry urgent data.

TCP implementations that queue urgent data for “out of band” processing should enforce per-connection limits on the amount of urgent data that they queue.

3.10. Options

[IANA, 2007] contains the official list of the assigned option numbers. [Hönes, 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 [Hönes, 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]

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

- Case 1: A single byte of **`option-kind`**.
- 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”.

There are a number of sanity checks that should be performed on TCP options before further option processing is done. These sanity checks help prevent a number of potential security problems, including buffer overflows. When these checks fail, the segment carrying the option should be silently dropped.

For options that belong to the “Case 2” described above, the following check should be performed:

`option-length` >= 2

> *The value “2” accounts for the* **`option-kind`** *byte and the* **`option-length`** *byte, and assumes zero bytes of option-data.*

This check prevents, among other things, loops in option processing that may arise from incorrect option lengths.

Additionally, while the option-length byte of TCP options of “Case 2” allows for an option length of up to 255 bytes, there is a limit on legitimate option length imposed by the syntax of the TCP header. Therefore, for all options of “Case 2”, the following check should be enforced:

option-offset + **`option-length`** <= **`Data Offset`** * 4

> *Where option-offset is the offset of the first byte of the option within the TCP header, with the first byte of the TCP header being assigned an offset of 0.*

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

The aforementioned check is meant to detect forged **`option-length`** values that might make an option overlap with the TCP payload, or even go past the actual end of the TCP segment carrying the option.

Section 3.1 of RFC 793 [Postel, 1981c] states that TCP must implement all the TCP options defined in that document. Additionally, a TCP implementation may support TCP extensions based on other TCP options as it sees fit, or as required by other specifications.

> *TCP Options have been specified in the past both within the IETF and by other groups.*

TCP must ignore unknown TCP options, provided they pass the validation checks described earlier in this Section. In the same way, middle-boxes such as packet filters should not reject TCP segments containing “unknown” TCP options that pass the validation checks described earlier in this Section.

> *There is renewed interest in defining new TCP options for purposes like improved connection management and maintenance, advanced congestion control schemes, and security features. The evolution of the TCP/IP protocol suite would be severely impacted by obstacles to deploying such new protocol mechanisms.*

In the past, TCP enhancements based on TCP options regularly have specified the exchange of a specific "enabling" option during the initial SYN/SYN-ACK handshake. Due to the severely limited TCP option space which has already become a concern, it should be expected that future specifications might introduce new options not negotiated or enabled in this way. Therefore, middle-boxes such as packet filters should not reject TCP segments containing unknown options solely because these options have not been present in the SYN/SYN-ACK handshake.

The specification of particular TCP options may contain specific rules for the syntax and placement of these options. These can only be enforced by end systems implementing these options, and the relevant specifications must point out the necessary details and related security considerations, which must be followed by implementers.

Some TCP implementations have been known to “echo” unknown TCP options received in incoming segments. Here we stress that TCP must not “echo” in any way unknown TCP options received in inbound TCP segments.

> *This is at the foundation for the introduction of new TCP options, ensuring unambiguous behaviour of systems not supporting a new specification.*

Section 4 of this document analyses the security implications of common TCP options.

3.11. 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.

3.12. 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 is used to indicate the “end of options” in those cases in which the end of options would not coincide with the end of the TCP header.

TCP implementations are required to ignore those options they do not implement, and to be able to handle options with illegal lengths. Therefore, TCP implementations should be able to gracefully handle those TCP segments in which the End of Option List should have been present, but is missing.

It is interesting to note that some TCP implementations do not use the “End of Option List” option for indicating the “end of options”, but simply pad the TCP header with several “No Operation” (Kind = 1) options to meet the header length specified by the **`Data Offset`** header field.

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 advertised maximum segment size may be the result of the consideration of a number of factors. Firstly, if fragmentation is employed, the size of the IP reassembly buffer may impose a limit on the maximum TCP segment size that can be received. Considering that the minimum IP reassembly buffer size is 576 bytes, if an MSS option is not present included in the connection-establishment phase, an MSS of 536 bytes should be assumed. Secondly, if Path-MTU Discovery (specified in RFC 1191 [Mogul and Deering, 1990] and RFC 1981 [McCann et al, 1996]) is expected to be used for the connection, an artificial maximum segment size may be enforced by a TCP to prevent the remote peer from sending TCP segments which would be too large to be transmitted without fragmentation. Finally, a system connected by a low-speed link may choose to introduce an artificial maximum segment size to enforce an upper limit on the network latency that would otherwise negatively affect its interactive applications [Stevens, 1994].

The option begins with an **`option-kind`** byte which must be equal to 2. It is followed by an option-length byte which must be equal to 4, and a two-byte field that holds the actual “maximum segment size”.

As stated in Section 3.1 of RFC 793 [Postel, 1981c], this option can only be sent in the initial connection request (i.e., in segments with the SYN control bit set). Therefore, the following check should be enforced on a TCP segment that carries this option:

`SYN` == 1

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

Given the option syntax, the option length must be equal to 4. Therefore, the following check should be performed:

`option-length` == 4

If the check fails, the TCP segment should be silently dropped.

The TCP specifications do not impose any requirements on the maximum segment size value that is included in the MSS option. However, there are a number of values that may cause undesirable results. Firstly, an MSS of 0 could possible “freeze” the TCP connection, as it would not allow data to be included in the payload of the TCP segments. Secondly, low values other than 0 would degrade the performance of the TCP connection (wasting more bandwidth in protocol headers than in actual data), and could potentially exhaust processing cycles at the sending TCP and/or the receiving TCP by producing an increase in the interrupt rate caused by the transmitted (or received) packets.

> *The problems that might arise from low MSS values were first described by [Reed, 2001]. However, the community did not reach consensus on how to deal with these issues at that point.*

RFC 791 [Postel, 1981a] requires IP implementations to be able to receive IP datagrams of at least 576 bytes. Assuming an IPv4 header of 20 bytes, and a TCP header of 20 bytes, there should be room in each IP packet for 536 application data bytes. Therefore, the received MSS could be sanitized as follows:

Sanitized_MSS = max(MSS, 536)

This “sanitized” MSS value would then be used to compute the “effective send MSS” by the expression included in Section 4.2.2.6 of RFC 1122 [Braden, 1989], as follows:

Eff.snd.MSS = min(Sanitized_MSS+20, MMS_S) - TCPhdrsize - IPoptionsize

where:

- Sanitized_MSS: Is the sanitized MSS value (the value received in the MSS option, with an enforced minimum value)
- MMS_S is the maximum size for a transport-layer message that TCP may send
- TCPhdrsize is the size of the TCP header, which typically was 20, but may be larger if TCP options are to be sent.
- IPoptionsize is the size of any IP options that TCP will pass to the IP layer with the current message.

There are two cases to analyse when considering the possible interoperability impact of sanitizing the received MSS value: TCP connections relying on IP fragmentation and TCP connections implementing Path-MTU Discovery. In case the corresponding TCP connection relies on IP fragmentation, given that the minimum reassembly buffer size is required to be 576 bytes by RFC 791 [Postel, 1981a], the adoption of 536 bytes as a lower limit is safe.

In case the TCP connection relies on Path-MTU Discovery, imposing a lower limit on the adopted MSS may ignore the advice of the remote TCP on the maximum segment size that can possibly be transmitted without fragmentation. As a result, this could lead to the first TCP data segment to be larger than the Path-MTU. However, in such a scenario, the TCP segment should elicit an ICMP Unreachable “fragmentation needed and DF bit set” error message that would cause the “effective send MSS” (E_MSS) to be decreased appropriately. Thus, imposing a lower limit on the accepted MSS will not cause any interoperability problems.

> *A possible scenario exists in which the proposed enforcement of a lower limit in the received MSS might lead to an interoperability problem. If a system was attached to the network by means of a link with an MTU of less than 576 bytes, and there was some intermediate system which either silently dropped (i.e., without sending an ICMP error message) those packets equal to or larger than that 576 bytes, or some intermediate system simply filtered ICMP “fragmentation needed and DF bit set” error messages, the proposed behavior would not lead to an interoperability problem, when communication could have otherwise succeeded. However, the interoperability problem would really be introduced by the network setup (e.g., the middle-box silently dropping packets), rather than by the mechanism proposed in this section. In any case, TCP should nevertheless implement a mechanism such as that specified by RFC 4821 [Mathis and Heffner, 2007] to deal with this type of “network black-holes”.*

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)

The SACK-permitted option is meant to advertise that the TCP sending this segment supports Selective Acknowledgements. The SACK-permitted option can be sent only in SYN segments. Therefore, the following check should be performed on TCP segments that contain this option:

`SYN` == 1

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

The SACK-permitted option is composed by an **`option-kind`** octet (which must be 4), and an **`option-length`** octet which must be 2. Therefore, the following check should be performed on the option:

`option-length` == 2

If the option does not pass this check, the TCP segment carrying the option should be silently dropped.

4.4.2. SACK Option (Kind = 5)

The SACK option is used to convey extended acknowledgment information from the receiver to the sender over an established TCP connection.

The option consists of an **`option-kind`** byte (which must be 5), an **`option-length`** byte, and a variable number of SACK blocks. Given that the space in the TCP header is limited, the following check should be enforced on the option field:

option-offset + **`option-length`** <= **`Data Offset`** * 4

If the option does not pass this check, the TCP carrying the option should be silently dropped.

A SACK Option with zero SACK blocks is nonsensical. Therefore, the following check should be performed:

`option-length` >= 10

> *The value “10” accounts for the option-kind byte, the option-length byte, a 4-byte left-edge field, and a 4-byte right-edge field.*

Furthermore, as stated in Section 3 of RFC 2018 [Mathis et al, 1996], a SACK option that specifies *n* blocks will have a length of 8*n*+2. Therefore, the following check should be performed:

(**`option-length`** - 2) % 8 == 0

If the **`option-length`** field does not pass this check, the TCP segment carrying the option should be silently dropped.

Each block included in a SACK option represents a number of received data bytes that are contiguous and isolated; that is, the bytes just below the block, (Left Edge of Block - 1), and just above the block, (Right Edge of Block), have not yet been received.

For obvious reasons, for each block included in the **`option-data`**, the following check should be enforced:

Left Edge of Block < Right Edge of Block

> *As in all the other occurrences in this document, all comparisons between sequence numbers should be performed using sequence number arithmetic.*

If any block contained in the option does not pass this check, the TCP segment should be silently dropped.

Potential of resource-exhaustion attacks

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.*

Therefore, it is clear that a limit should be imposed on the number of SACK blocks that a TCP will store in memory for each connection at any time. Measurements in [Dharmapurikar and Paxson, 2005] indicate that in the vast majority of cases connections have a single hole in the data stream at any given time. Thus, a limit of 16 SACK blocks for each connection would handle even most of the more unusual cases in which there is more than one simultaneous hole at a time.

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.

As with all TCP options of “Case 2”, the following check should be enforced on the option-length field:

option-offset + **`option-length`** <= **`Data Offset`** * 4

If the option does not pass this check, the TCP segment carrying the option should be silently dropped.

Given that the MD5 has a fixed length, the following check should be performed on the MD5 option:

`option-length` == 18

If the option does not pass this check, the TCP segment containing the option should be silently dropped.

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.*

There are a number of ongoing efforts within the IETF to develop a replacement for the address the weaknesses of the basic TCP MD5 option. Some of them aim at completely replacing the TCP MD5 option, while others aim at improving the current option by, for example, standardising mechanisms for re-keying.

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.

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.

> *[Welzl, 2008] describes major problems with the use of the Window scale option in the Internet due to faulty equipment.*

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`**).

The option may be sent only in the initial SYN segment, but may also be sent in a SYN/ACK segment if the option was received in the initial SYN segment. If the option is received in any other segment, it should be silently dropped.

As discussed above, the option-length must be 3. Therefore, the following check should be enforced:

`option-length` == 3

If the option does not pass this check, the TCP segment carrying this option should be silently ignored.

As discussed in Section 2.3 of RFC 1323 [Jacobson et al, 1992], in order to prevent new data from being mistakenly considered as old and vice versa, the resulting window should be equal to or smaller than 2^32. Therefore, an upper limit should be enforced on the shift count (**`shift.cnt`**):

`shift.cnt` <= 14

If the option does not pass this check, the option-data should be set to 14.

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). As defined by RFC 1323, the option-length must be 10. Therefore, the following check should be enforced:

`option-length` == 10

If the option does not pass this check, the TCP segment carrying the option should be silently dropped.

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.

We therefore recommend that timestamps are generated with a similar algorithm to that introduced by RFC 1948 [Bellovin, 1996] for the generation of Initial Sequence Numbers (ISNs). That is:

timestamp = T() + F(localhost, localport, remotehost, remoteport, secret_key)

where the result of *T()* is a global system clock that complies with the requirements of Section 4.2.2 of RFC 1323 [Jacobson et al, 1992], and *F()* is a function that should not be computable from the outside. Therefore, we suggest *F()* to be a cryptographic hash function of the connection-id and some secret data.

F() provides an offset that will be the same for all incarnations of a connection between the same two endpoints, while *T()* provides the monotonically increasing values that are needed for PAWS.

> *[Gont, 2008c] is CPNI's effort at the IETF to document this recommended scheme for generating TCP timestamps.*

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 initialise 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 [Postel, 1981c] 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 [Postel, 1981c].*

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 [Postel, 1981c].*

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-synchronised 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-synchronised 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:

- 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).
- 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 be in the ESTABLISHED state on the client side, but in the CLOSED state at the server side. 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.
- 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.
- 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-synchronised 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.

[Zúquete, 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.

```
     TCP A                                                TCP B

1. CLOSED                                                 LISTEN
2. SYN-SENT     --> <SEQ=100><CTL=SYN>                --> LISTEN
3. SYN-RECEIVED <-- <SEQ=300><CTL=SYN>                <-- LISTEN
4. SYN-RECEIVED --> <SEQ=100><ACK=301><CTL=SYN,ACK>   --> ESTABLISHED
5. ESTABLISHED  <-- <SEQ=301><ACK=101><CTL=SYN,ACK>   <-- ESTABLISHED
6. ESTABLISHED  --> <SEQ=101><ACK=301><CTL=ACK><DATA> --> ESTABLISHED
```

Figure 5: 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:

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

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.

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

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).

Enforcing per-user and per-process limits

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.

Limiting the number of simultaneous connections at the application

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).

Limiting the number of simultaneous connections at firewalls

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

Some firewalls block incoming TCP connections by blocking only incoming SYN segments. However, there are inconsistencies in how different TCP implementations handle SYN segments that have additional flags set, which may allow an attacker to bypass firewall rules [US-CERT, 2003b].

For example, some firewalls have been known to mistakenly allow incoming SYN segments if they also have the RST bit set. As some TCP implementations will create a new connection in response to a TCP segment with both the **SYN** and **RST** bits set, an attacker could bypass the firewall rules and establish a connection with a “protected” system by setting the **RST** bit in his SYN segments.

Here we advise TCP implementations to silently drop those TCP segments that have both the **SYN** and the **RST** flags set.

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.

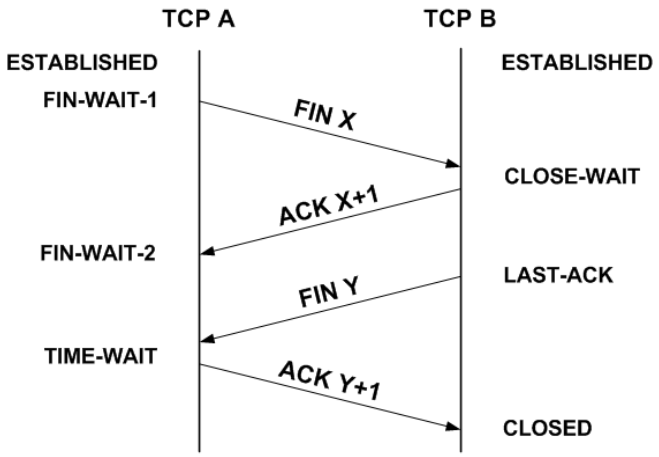

Figure 6: TCP connection-termination mechanism

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 [Postel, 1981c] 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.

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 limits on the duration of the FIN-WAIT-2 state

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.

We advise the implementation of such a timeout for the FIN-WAIT-2 state.

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:

- A call to shutdown(sockfd, SHUT_WR), to close the sending direction of this connection.
- Successive calls to read(), until it returns 0, thus indicating that the remote TCP peer has finished sending data.
- 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).
- 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:

- 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).
- 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.

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:

- 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 specialised 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.
- 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.
- 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.
- 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.
- 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:

- A call to shutdown(sockfd, SHUT_WR), to close the sending direction of this connection
- Successive calls to read(), until it returns 0, thus indicating that the remote TCP peer has finished sending data.
- 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).
- 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:

- 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).
- 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:

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

7.3.1. Automatic send-buffer tuning mechanisms

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.

As a result, the following limit should be enforced on the size of each send buffer:

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)

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.

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.

As a result, the following limit should be enforced on the size of each receive buffer:

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).

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 [Postel, 1981c], 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 normalise 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

TCP implements two algorithms, “slow start” and “congestion avoidance”, for controlling the rate at which data is transmitted on a TCP connection [Allman et al, 1999]. These algorithms require the addition of two variables as part of TCP per-connection state: *cwnd* and *ssthresh*.

The congestion window (*cwnd*) is a sender-side limit on the amount of outstanding data that the sender can have at any time, while the receiver’s advertised window (*rwnd*) is a receiver-side limit on the amount of outstanding data. The minimum of *cwnd* and *rwnd* governs data transmission.

Another state variable, the slow-start threshold (*ssthresh*), is used to determine whether it is the slow start or the congestion avoidance algorithm that should control data transmission. When *cwnd* < *ssthresh*, “slow start” governs data transmission, and the congestion window (*cwnd*) is exponentially increased. When *cwnd* > *ssthresh*, “congestion avoidance” governs data transmission, and the congestion window (*cwnd*) is only linearly increased.

> *As specified in RFC 2581 [Allman et al, 1999], when cwnd and ssthresh are equal the sender may use either slow start or congestion avoidance.*

During slow start, TCP increments *cwnd* by at most *SMSS* bytes for each ACK received that acknowledges new data. During congestion avoidance, *cwnd* is incremented by 1 full-sized segment per round-trip time (RTT), until congestion is detected.

Additionally, TCP uses two algorithms, Fast Retransmit and Fast Recovery, to mitigate the effects of packet loss. The “Fast Retransmit” algorithm infers packet loss when three Duplicate Acknowledgements (DupACKs) are received.

> *The value “three” is meant to allow for fast-retransmission of “missing” data, while avoiding network packet reordering from triggering loss recovery.*

Once packet loss is detected by the receipt of three duplicate-ACKs, the “Fast Recovery” algorithm governs the transfer of new data until a non-duplicate ACK is received that acknowledges the receipt of new data. The Fast Retransmit and Fast Recovery algorithms are usually implemented together, as follows (from RFC 2581):

- When the third duplicate ACK is received, set ssthresh to no more than the value given in the equation: ssthresh = max (FlightSize / 2, 2*SMSS)
- Retransmit the lost segment and set cwnd to ssthresh plus 3*SMSS. This artificially "inflates" the congestion window by the number of segments (three) that have left the network and which the receiver has buffered.

- For each additional duplicate ACK received, increment cwnd by SMSS. This artificially inflates the congestion window in order to reflect the additional segment that has left the network.
- Transmit a segment, if allowed by the new value of cwnd and the receiver's advertised window.
- When the next ACK arrives that acknowledges new data, set cwnd to ssthresh (the value set in step 1). This is termed "deflating" the window.

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.

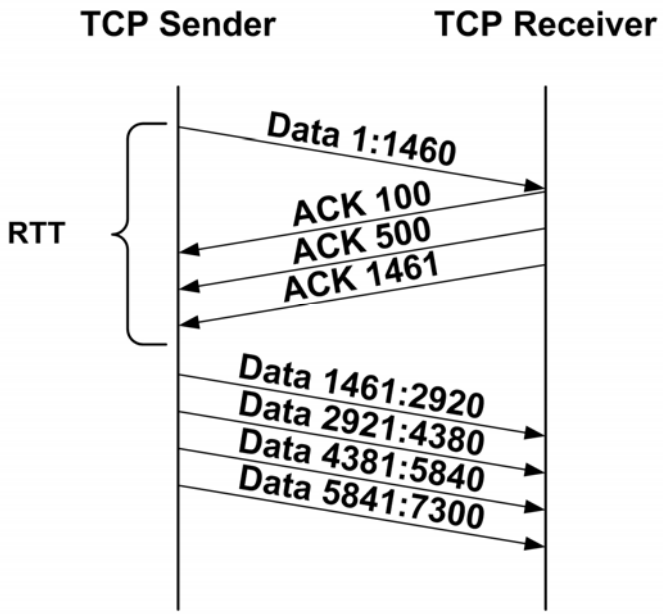

Figure 7: ACK division attack

[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.

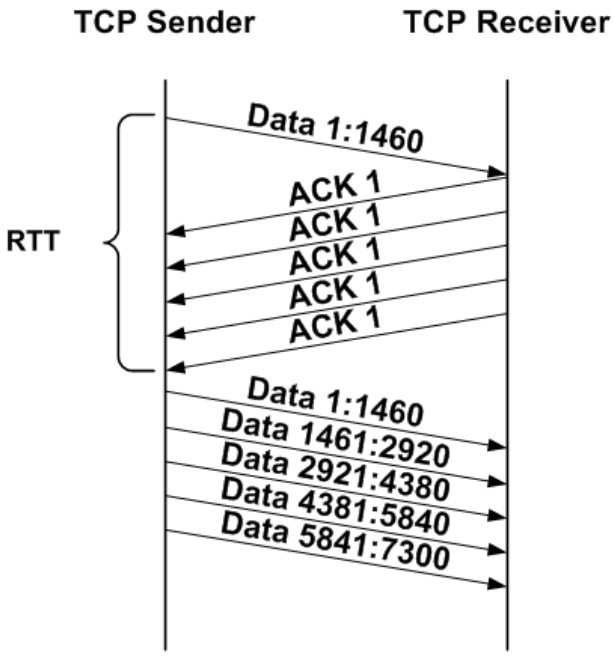

Figure 8: DupACK forgery attack

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.

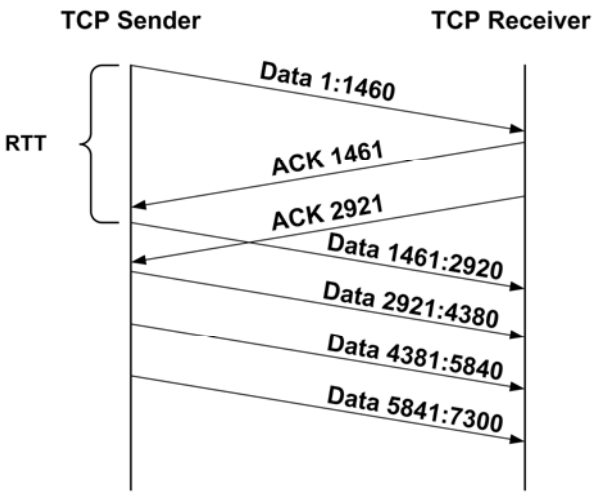

Figure 9: Optimistic ACKing attack

[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 penalise a TCP receiver that acknowledges data not yet sent by resetting the corresponding connection. Here we deprecate 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 2581 [Allman et al, 1999], 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 [Postel, 1981c] 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, 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.

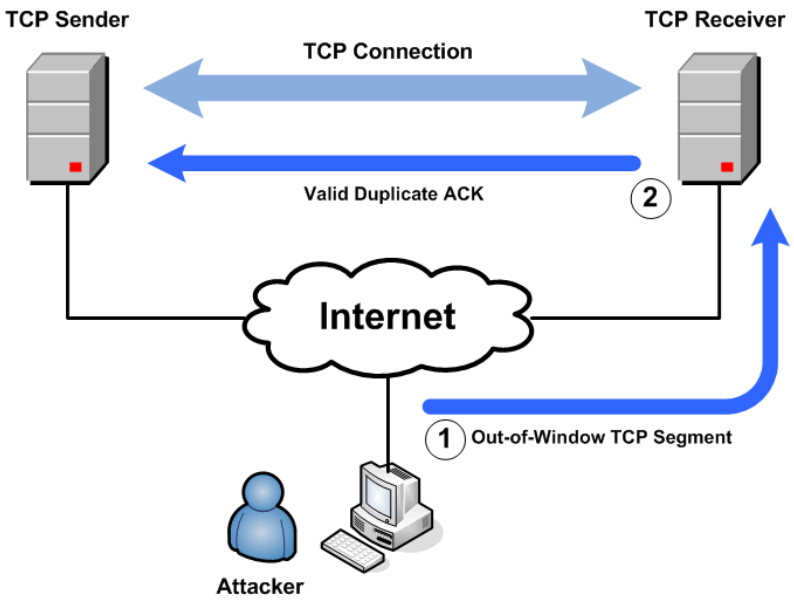

Figure 10: Blind Dup-ACK forgery attack

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.

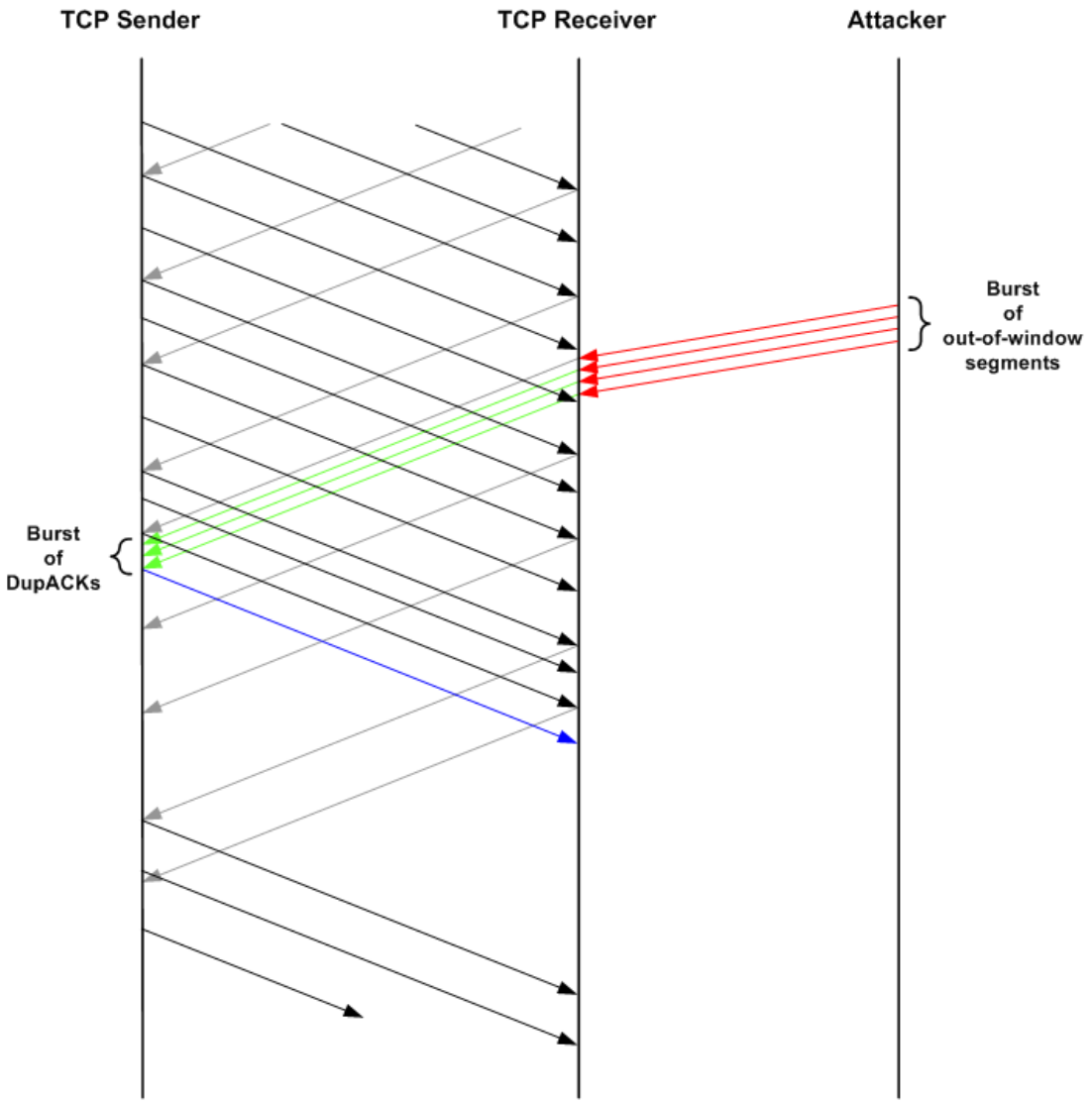

Figure 11: Blind throughput-reduction attack (time-line graph)

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.

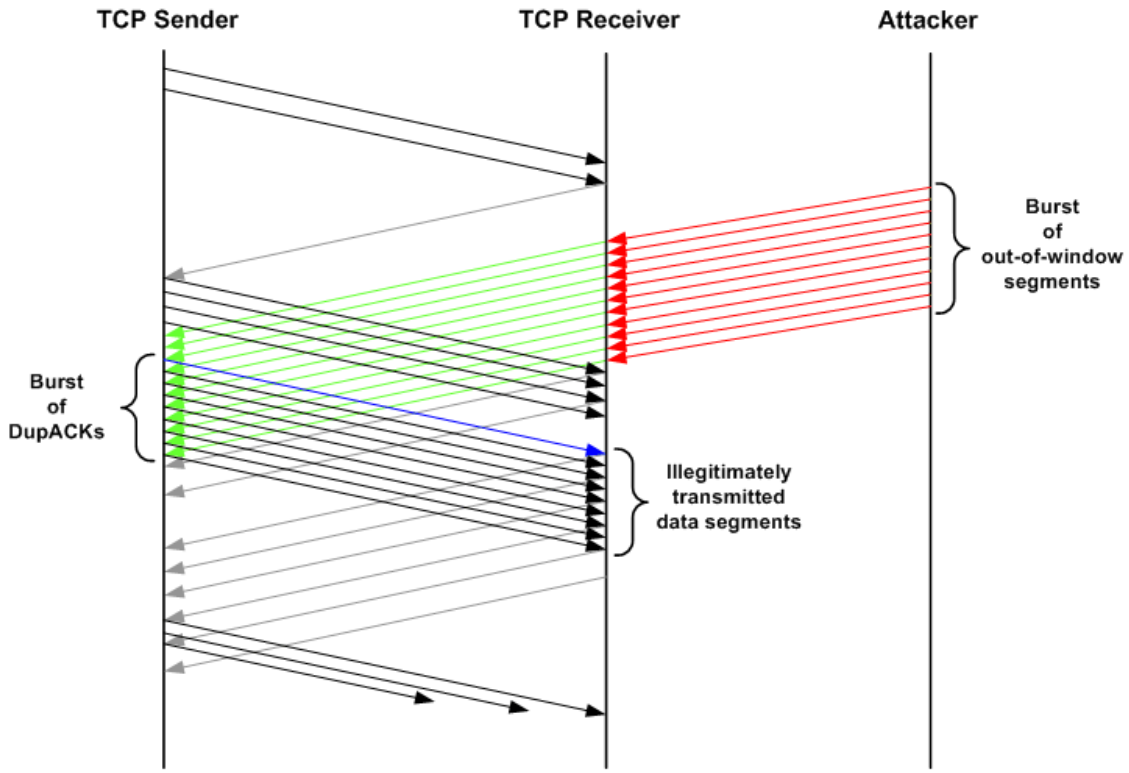

Figure 12: Blind flooding attack (time-line graph)

Figure 13 is a time-sequence graph produced from packet logs obtained from tests of the described attack in a real network. A burst of segments is sent upon receipt of the burst of Duplicate Acknowledgements illegitimately elicited by the attacker. Figure 14 is an averaged-throughput graphic for the same time frame, which clearly shows the effect of the attack in terms of throughput.

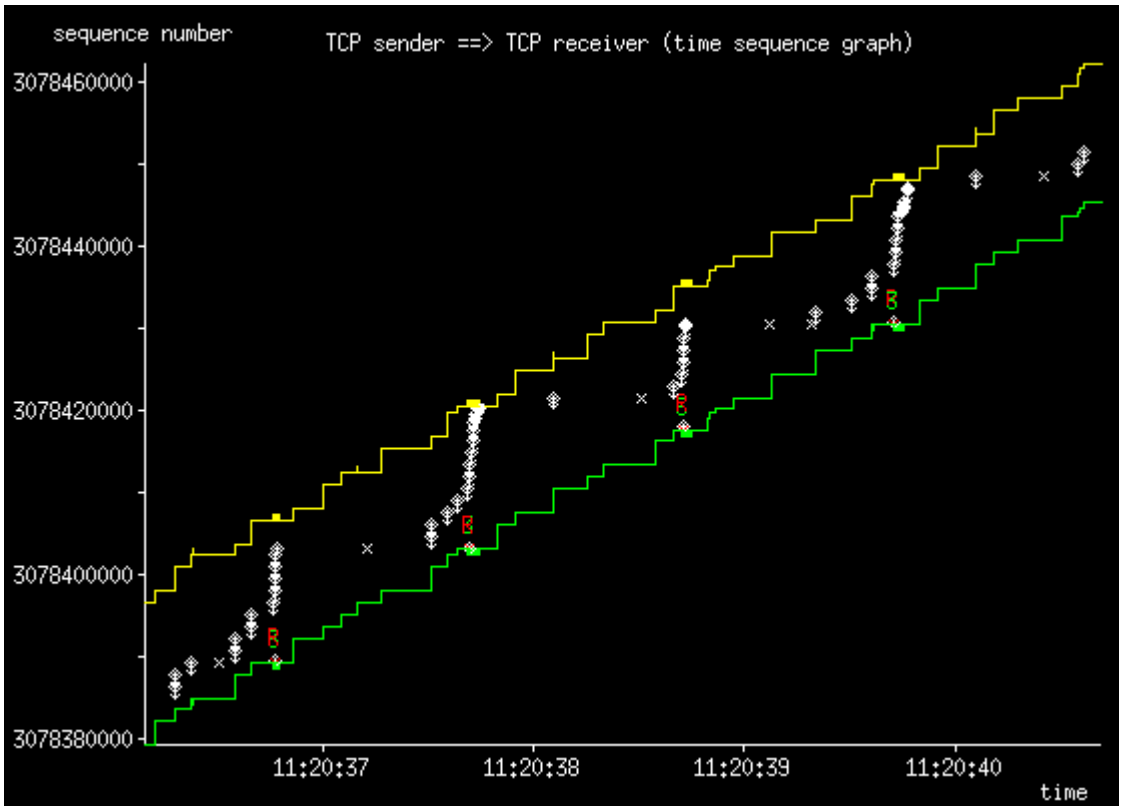

Figure 13: Blind flooding attack (time sequence graph)

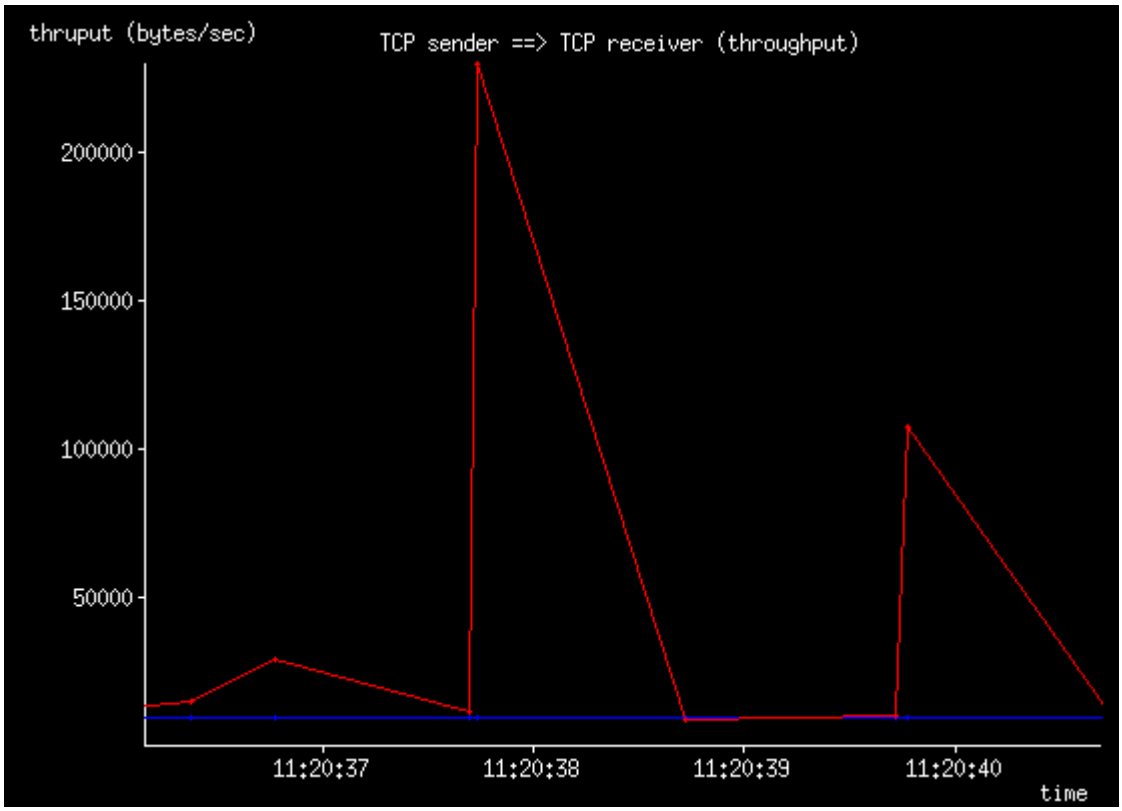

Figure 14: Blind flooding attack (averaged throughput graph)

These graphics were produced with Shawn Ostermann's tcptrace tool [Ostermann, 2008]. An explanation of the format of the graphics can be found in tcptrace's manual (available at the project's web site: http://www.tcptrace.org).

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 defence 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 fulfil, 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.

Figure 15 is a time-sequence graph produced from packet logs obtained from tests performed in a real network. Once loss recovery is illegitimately triggered by the duplicate-ACKs elicited by the attacker, an entire flight of data is unnecessarily retransmitted. Figure 16 is an averaged-throughput graphic for the same time-frame, which shows an increase in the throughput of the connection resulting from the retransmission of segments governed by NewReno's loss recovery.

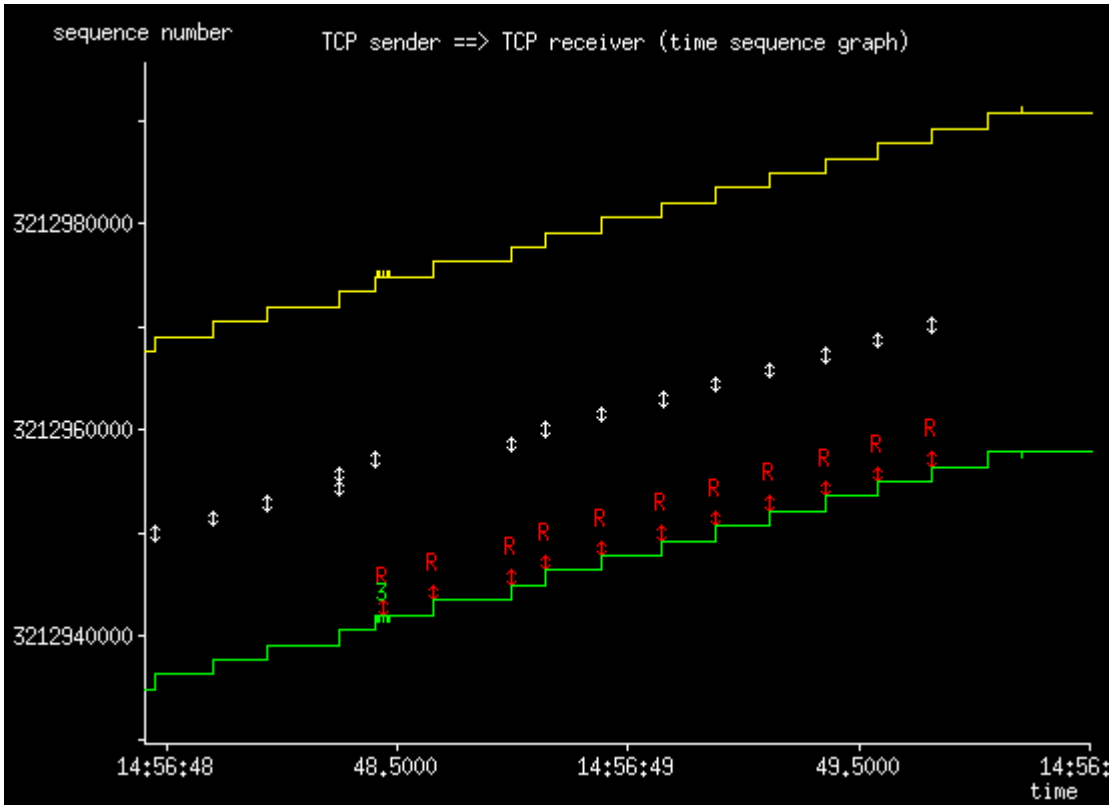

Figure 15: NewReno loss recovery (time-sequence graph)

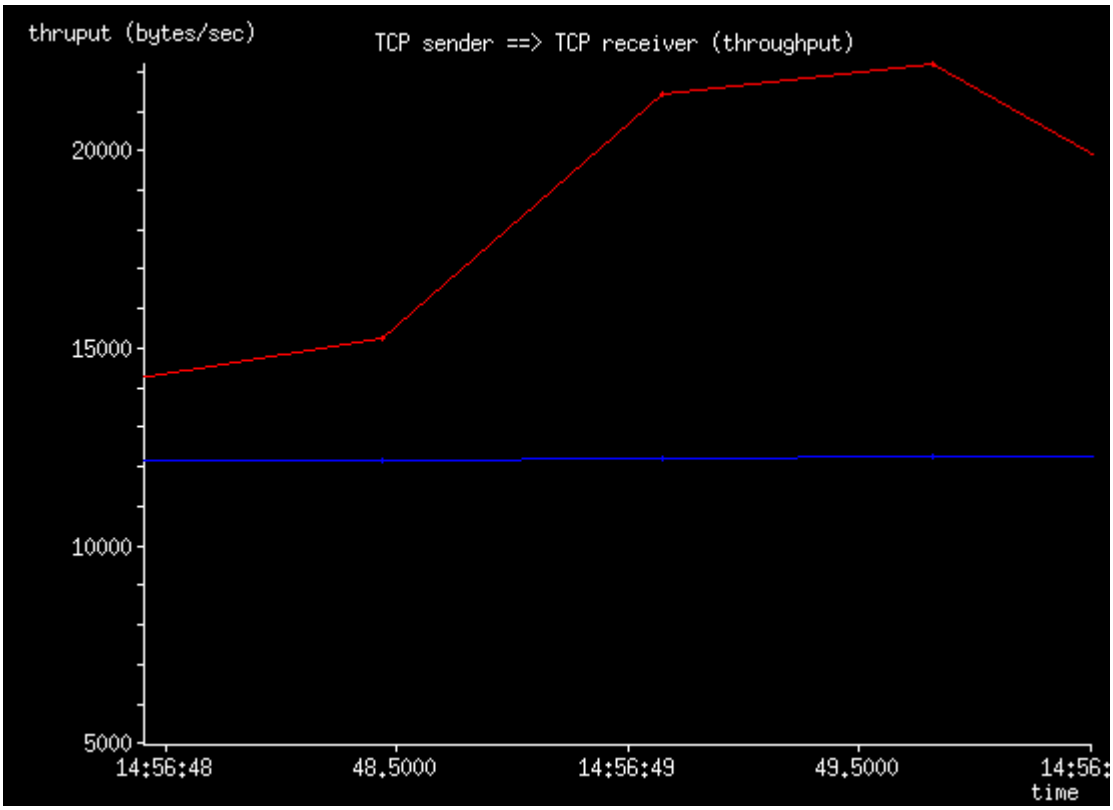

Figure 16: NewReno loss recovery (averaged throughput graph)

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

RFC 3517 [Blanton et al, 2003] 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 [Allman et al, 1999]. 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. However, as RFC 3517 does not actually require 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 may still remain vulnerable to the blind throughput-reduction attack described in Section 9.2.1. SACK-based loss recovery implementations

should be updated to implement the countermeasure (“Use of SACK information to validate DupACKs”) described in Section 9.2.5.

9.2.5. Countermeasures

Validating TCP sequence numbers

As discussed in Section 9.2, TCP responds with an ACK when an out-of-window segment is received, to accommodate those scenarios in which the Acknowledgement segments that correspond to some received data are lost in the network, and to help discover half-open TCP connections.

However, it is possible to restrict the sequence numbers that are considered acceptable, and have TCP respond with ACKs only when it is strictly necessary.

The following check could be performed on the TCP sequence number of an incoming TCP segment:

$$RCV.NXT - MAX.RCV.WND <= SEG.SEQ <= RCV.NXT + RCV.WND$$

Equation 2: Validating TCP Sequence Numbers

where MAX.RCV.WND is the largest TCP window that has so far been advertised to the remote endpoint.

If a segment passes this check, the processing rules specified in RFC 793 [Postel, 1981c] should be applied. Otherwise, TCP should send an ACK (as specified by the processing rules in RFC 793 [Postel, 1981c]), applying rate-limiting to the Acknowledgement segments sent in response to out-of-window segments.

Discussion

A feature of TCP is that, in some scenarios, it can detect half-open connections. If an implementation chose to silently drop those TCP segments that do not pass the check enforced by Equation 2, it could prevent TCP from detecting half-open connections. Figure 17 shows a scenario in which, provided that “TCP B” behaves as specified in RFC 793, a half-open connection would be discovered and aborted.

> *An established connection is said to be “half open” if one of the TCPs has closed or aborted the connection at its end without the knowledge of the other, or if the two ends of the connection have become desynchronised owing to a crash that resulted in loss of memory.*

```
    TCP A                                                TCP B

1.  (CRASH)                                  (send 300,receive 100)

2.  CLOSED                                               ESTABLISHED

3.  SYN-SENT --> <SEQ=400><CTL=SYN>               --> (??)

4.  (!!)     <-- <SEQ=300><ACK=100><CTL=ACK>     <-- ESTABLISHED

5.  SYN-SENT --> <SEQ=100><CTL=RST>               --> (Abort!!)
```

Figure 17: Half-Open Connection Discovery

In the scenario illustrated by Figure 17, TCP A crashes losing the connection-state information of the TCP connection with TCP B. In line 3, TCP A tries to establish a new connection with TCP B, using the same four-tuple {IP Source Address, TCP source port, IP Destination Address, TCP destination port}. In line 4, as the SYN segment is out of window, TCP B responds with an ACK. This ACK elicits an RST segment from TCP A, which causes the half-open connection at TCP B to be aborted.

> *If the SYN segment had been "in window", TCP B would have sent an RST segment instead, which would have closed the half-open connection. Ongoing work at the TCPM WG of the IETF proposes to change this behavior, and make TCP respond to a SYN segment received for any of the synchronised states with an ACK segment, to avoid in-window SYN segments from being used to perform connection-reset attacks [Ramaiah et al, 2008].*

However, in case the out-of-window segment was silently dropped, the scenario in Figure 17 would change into that in Figure 18.

```
    TCP A                                                TCP B

1.  (CRASH)                                  (send 300,receive 100)

2.  CLOSED                                               ESTABLISHED

3.  SYN-SENT --> <SEQ=400><CTL=SYN>               -->

                         ........
```

Figure 18: Half-Open Connection Discovery with the proposed counter-measure

In line 3, the SYN segment sent by TCP A is silently dropped by TCP B because it does not pass the check enforced by Equation 2 (i.e., it contains an out-of-window sequence number). As a result, some time later (an RTO) TCP A retransmits its SYN segment. Even after TCP A times out, the half-open connection at TCP B will remain in the same state.

Thus, a conservative reaction to those segments that do not pass the check enforced by Equation 2 would be to respond with an Acknowledgement segment (as specified by RFC 793), applying rate-limiting to those Acknowledgement segments sent in response to segments that do not pass the check enforced by that equation. An implementation might choose to enforce a rate-limit of, e.g., one ACK per five seconds, as a single ACK segment is needed for the Half-Open Connection Discovery mechanism to work.

> *As the only reason to respond with an ACK to those segments that do not pass the check enforced by Equation 2 is to allow TCP to discover half-open connections, an aggressive rate-limit can be enforced. As long as the rate-limit prevents out-of-window segments from eliciting three Acknowledgment segments in a Round-trip Time (RTT), an attacker would not be able to trigger TCP's loss-recovery, and thus would not be able to perform the attacks described in the previous sections.*

It is interesting to note that RFC 793 [Postel, 1981c] itself states that half-open connections are expected to be unusual. Additionally, given that in many scenarios it may be unlikely for a TCP connection request to be issued with the same four-tuple as that of the half-open connection, a complete solution for the discovery of half-open connections cannot rely on the mechanism illustrated by Figure 17, either. Therefore, some implementations might choose to sacrifice TCP's ability to detect half-open connections, and have a more aggressive reaction to those segments that do not pass the check enforced by Equation 2 by silently dropping them.

> *This validation check can also help to avoid ACK wars in some scenarios that may arise from the use of transparent proxies. In those scenarios, when the transparent proxy fails to wire (i.e., is disabled), the sequence numbers of the two end-points of the TCP connection become desynchronised, and both TCPs begin to send duplicate Acknowledgements to each other, with the intention of re-synchronising them. As the sequence numbers never get re-synchronised, the ACK war can only be stopped by an external agent.*

Limiting the number of duplicate acknowledgments

Given that duplicate acknowledgements should be elicited by out-of-order segments, a TCP sender could limit the number of duplicate acknowledgements it will honour to:

$$\text{Max\_DupACKs} = (\mathit{FlightSize} / \mathit{SMSS}) - 1$$

Where *FlightSize* and *SMSS* are the values defined in RFC 2581 [Allman et al, 1999]. When more than Max_DupACKs duplicate acknowledgements are received, the exceeding DupACKs should be silently dropped.

Use of SACK information to validate DupACKs

SACK, specified in 2018 [Mathis et al, 1996], provides a mechanism for TCP to be able to acknowledge the receipt of out-of-order TCP segments. For connections that have agreed to use SACK, each legitimate DupACK will contain new SACK information that reflects the data bytes contained in the out-of-order data segment that elicited the DupACK.

RFC 3517 [Blanton et al, 2003] specifies a SACK-based loss recovery algorithm for TCP. However, it does recommend TCP implementations to validate DupACKs by requiring that they contain new SACK information. Results obtained from auditing a number of TCP implementations seem to indicate that most TCP implementations do not enforce this validation check on incoming DupACKs, either.

In the case of TCP connections that have agreed to use SACK, a validation check should be performed on incoming ACK segments to completely eliminate the attacks described in Section 9.2.1 and Section 9.2.2 of this document: “Duplicate ACKs should contain new SACK information. The SACK information should refer to data that has already been sent, but that has not yet been acknowledged by TCP’s cumulative Acknowledgement”.

Those ACK segments that do not comply with this validation check should not be considered “duplicate ACKs”, and thus should not trigger the loss-recovery phase.

In case at least one segment in a window of data has been lost, the successive segments will elicit the generation of Duplicate ACKs containing new SACK information. This SACK information will indicate the receipt of these successive segments by the TCP receiver.

In the case of pure ACKs illegitimately elicited by out-of-window segments, however, the ACKs will not contain any SACK information.

> *If DSACK (specified in 2883 [Floyd et al, 2000]) were implemented by the TCP receiver, then the illegitimately elicited DupACKs might contain* out-of-window *SACK information if the sequence number of the forged TCP segment (SEG.SEQ) is lower than the next expected sequence number (RECV.NXT) at the TCP receiver. Such segments should be considered to indicate the receipt of duplicate data, rather than an indication of lost data, and therefore should not trigger loss recovery.*

TCP port number randomisation

As in order to perform the blind attacks described in Section 9.2.1 and Section 9.2.2 the attacker needs to know the TCP port numbers in use by the connection to be attacked, obfuscating the TCP source port used for outgoing TCP connections will increase the number of packets required to successfully perform these attacks. Section 3.1 of this document discusses the use of port randomisation.

It must be noted that given that these blind DupACK triggering attacks do not require the attacker to forge valid TCP Sequence numbers and TCP Acknowledgement numbers, port randomisation should not be relied upon as a first line of defense.

Ingress and Egress filtering

Ingress and Egress filtering reduces the number of systems in the global Internet that can perform attacks that rely on forged source IP addresses. While protection from the blind attacks discussed in Section 9.2 should not rely only on Ingress and Egress filtering, its deployment is recommended to help prevent all attacks that rely on forged IP addresses. RFC 3704 [Baker and Savola, 2004], RFC 2827 [Ferguson and Senie, 2000], and [NISCC, 2006] provide advice on Ingress and Egress filtering.

Generalized TTL Security Mechanism (GTSM)

RFC 5082 [Gill et al, 2007] proposes a check on the TTL field of the IP packets that correspond to a given TCP connection to reduce the number of systems that could successfully attack the protected TCP connection. It provides for the attacks discussed in this document the same level of protection than for the attacks described in [Watson, 2004] and RFC 4953 [Touch, 2007]. While implementation of this mechanism may be useful in some scenarios, it should be clear that countermeasures discussed in the previous sections provide a more effective and simpler solution than that provided by the GTSM.

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]).*

The following subsections try to summarise the security implications of ECN.

9.3.1. Possible attacks by a compromised router

Firstly, a router controlled by a malicious user could erase the CE codepoint (either by replacing it with the ECT(0), ECT(1), or non-ECT codepoints), effectively eliminating the congestion indication. As a result, the corresponding TCP sender would not reduce its data transmission rate, possibly leading to network congestion. This could also lead to unfairness,

as this flow could experience better performance than other flows for which the congestion indication is not erased (and thus their transmission rate is reduced).

Secondly, a router controlled by a malicious user could illegitimately set the CE codepoint, falsely indicating congestion, to cause the TCP sender to reduce its data transmission rate. However, this particular attack is no worse than the malicious router simply dropping the packets rather setting their CE codepoint.

Thirdly, a malicious router could turn off the ECT codepoint of a packet, thus disabling ECN support. As a result, if the packet later arrives at a router that is experiencing congestion, it may be dropped rather than marked. As with the previous scenario, though, this is no worse than the malicious router simply dropping the corresponding packet.

It should be noted that a compromised on-path IP router could engage in a much broader range of attacks, with broader impacts, and at much lower attacker cost than the ones described here. Such a compromised router is extremely unlikely to engage in the attack vectors discussed in this section, given the existence of more effective attack vectors that have lower attacker cost.

9.3.2. Possible attacks by a malicious TCP endpoint

If a packet with the ECT codepoint set arrives at an ECN-capable router that is experiencing moderate congestion, the router may decide to set its CE codepoint instead of dropping it. If either of the TCP endpoints do not honour the congestion indication provided by an ECN-capable router, this would result in unfairness, as other (legitimate) ECN-capable flows would still reduce their sending rate in response to the ECN marking of packets. Furthermore, under moderate congestion, non-ECN-capable flows would be subject to packet drops by the same router. As a result, the flow with a malicious TCP end-point would obtain better service than the legitimate flows.

As noted in RFC 3168 [Ramakrishnan et al, 2001], a TCP endpoint falsely indicating ECN capability could lead to unfairness, allowing the mis-beheaving flow to get more than its fair share of the bandwidth. This could be the result of the mis-behavior of either of the TCP endpoints. For example, the sending TCP could indicate ECN capability, but then send a CWR in response to an ECE without actually reducing its congestion window. Alternatively (or in addition), the receiving TCP could simply ignore those packets with the CE codepoint set, thus avoiding the sending TCP from receiving the congestion indication.

In the case of the sending TCP ignoring the ECN congestion indication, this would be no worse than the sending TCP ignoring the congestion indication provided by a lost segment. However, the case of a TCP receiver ignoring the CE codepoint allows the TCP receiver to get more than its fair share of bandwidth in a way that was previously unavailable. If congestion was kept “moderate”, then the malicious TCP receiver could maintain the unfairness, as the router experiencing congestion would mark the offending packets of the misbehaving flow rather than dropping them. At the same time, legitimate ECN-capable flows would respond to the congestion indication provided by the CE codepoint, while legitimate non-ECN-capable flows would be subject of packet dropping. However, if congestion turned to sufficiently heavy, the router experiencing congestion would switch from marking packets to dropping packets,

and at that point the attack vector provided by ECN could no longer be exploited (until congestion returns to moderate state).

RFC 3168 [Ramakrishnan et al, 2001] 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

Section 3.8 of RFC 793 [Postel, 1981c] 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

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 [Postel, 1981c], 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.

In order to eliminate this vulnerability, when there is already a pending passive OPEN for some local port number, only processes belonging to the same user should be able to “reuse” the local port for another passive OPEN. Additionally, reuse of a local port could default to

“off”, and be enabled only by an explicit command (e.g., the *setsockopt()* function of the Sockets API).

10.2. Active opens and binding sockets

As discussed in Section 10.1, the “OPEN” command specified in Section 3.8 of RFC 793 [Postel, 1981c] 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

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 [Postel, 1981c], 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.

[Ramaiah et al, 2008] 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. With the implementation of the proposed mechanism, TCP would behave as follows:

If the **Sequence Number** of an RST segment is outside the receive window, the segment is silently dropped (as stated by RFC 793). That is, a reset segment is discarded unless it passes the following check:

RCV.NXT <= **Sequence Number** < RCV.NXT+RCV.WND

If the sequence number falls exactly on the left-edge of the receive window, the reset is honoured. That is, the connection is reset if the following condition is true:

Sequence Number == RCV.NXT

If an RST segment passes the first check (i.e., it is within the receive window) but does not pass the second check (i.e., it does not fall exactly on the left edge of the receive window), an Acknowledgement segment ("challenge ACK") is set in response:

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

This Acknowledgement segment is referred to as a "challenge ACK" as, in the event the RST segment that elicited it had been legitimate (but silently dropped as a result of enforcing the above checks), the challenge ACK would elicit a new reset segment that would fall exactly on the left edge of the window and would thus pass all the above checks, finally resetting the connection.

We recommend the implementation of this countermeasure. However, 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 [Postel, 1981c] 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.

The IETF has been working on a document, "Improving TCP's Resistance to Blind In-Window Attacks" [Ramaiah et al, 2008] which addresses, among others, this variant of TCP-based connection-reset attack. This section describes the counter-measure proposed by the IETF, a problem that may arise from the implementation of that solution, and a workaround to it.

In order to mitigate this attack vector, [Ramaiah et al, 2008] proposes to change TCP's reaction to SYN segments as follows. When a SYN segment is received for a connection in any of the synchronised states, an Acknowledgement (ACK) segment is sent in response.

> *As discussed in [Ramaiah et al, 2008], there is a corner-case that would not be properly handled by this mechanism. If a host (TCP A) establishes a TCP connection with a remote peer (TCP B), and then crashes, reboots and tries to initiate a new incarnation of the same connection (i.e., a connection with the same four-tuple as the previous connection) using an Initial Sequence Number equal to the RCV.NXT value at the remote peer (TCP B), the ACK segment sent by TCP B in response to the SYN segment would contain an Acknowledgement number that would be considered valid by TCP A, and thus an RST segment would not be sent in response to the Acknowledgement (ACK) segment. As this ACK would not have the SYN bit set, TCP A (being in the SYN-SENT state) would silently drop it (as stated on page 68 of RFC 793). After a Retransmission Timeout (RTO), TCP A would retransmit its SYN segment, which would lead to the same sequence of events as before. Eventually, TCP A would timeout, and the connection would be aborted. This is a corner case in which the introduced change would lead to a non-desirable behavior. However, we consider this scenario to be extremely unlikely and, in the event it ever took place, the connection would nevertheless be aborted after retrying for a period of USER TIMEOUT seconds.*

However, when this change is implemented exactly as described in [Ramaiah et al, 2008], the potential of interoperability problems is introduced, as a heuristic widely incorporated in many TCP implementations is disabled.

In a number of scenarios a socket pair may need to be reused while the corresponding four-tuple is still in the TIME-WAIT state in a remote TCP peer. For example, a client accessing some service on a host may try to create a new incarnation of a previous connection, while the corresponding four-tuple is still in the TIME-WAIT state at the remote TCP peer (the server). This may happen if the ephemeral port numbers are being reused too quickly, either because of a bad policy of selection of ephemeral ports, or simply because of a high connection rate to the corresponding service. In such scenarios, the establishment of new connections that reuse a four-tuple that is in the TIME-WAIT state would fail. In order to avoid this problem, RFC 1122 [Braden, 1989] states (in Section 4.2.2.13) that when a connection request is received with a four-tuple that is in the TIME-WAIT state, the connection request could be accepted if the sequence number of the incoming SYN segment is greater than the last sequence number seen on the previous incarnation of the connection (for that direction of the data transfer).

> *This requirement aims at avoiding the sequence number space of the new and old incarnations of the connection to overlap, thus avoiding old segments from the previous incarnation of the connection to be accepted as valid by the new connection.*

The requirement in [Ramaiah et al, 2008] to disregard SYN segments received for connections in any of the synchronised states forbids the implementation of the heuristic described above. As a result, we argue that the processing of SYN segments proposed in

[Ramaiah et al, 2008] should apply only for connections in any of the synchronized states other than the TIME-WAIT state.

The following paragraphs summarize the processing of SYN segments in the synchronized states, such that connection-reset attacks are mitigated, while interoperability is not affected. Additionally, the timestamp option of the incoming SYN segment is included (if present) in the heuristics performed for allowing a high connection-establishment rate, thus improving the robustness of TCP.

Processing of SYN segments received for connections in the synchronized states should occur as follows:

- If a SYN segment is received for a connection in any synchronized state other than TIME-WAIT, respond with an ACK, applying rate-throttling.
- If the corresponding connection is in the TIME-WAIT state, then,
- If the previous incarnation of the connection used timestamps, then,
- If TCP timestamps would be enabled for the new incarnation of the connection, and the timestamp contained in the incoming SYN segment is greater than the last timestamp seen on the previous incarnation of the connection (for that direction of the data transfer), honour the connection request (creating a connection in the SYN-RECEIVED state).
- If TCP timestamps would be enabled for the new incarnation of the connection, the timestamp contained in the incoming SYN segment is equal to the last timestamp seen on the previous incarnation of the connection (for that direction of the data transfer), and the Sequence Number of the incoming SYN segment is larger than the last sequence number seen on the previous incarnation of the connection (for that direction of the data transfer), then honour the connection request (creating a connection in the SYN-RECEIVED state).
- If TCP timestamps would not be enabled for the new incarnation of the connection, but the Sequence Number of the incoming SYN segment is larger than the last sequence number seen on the previous incarnation of the connection (for the same direction of the data transfer), honour the connection request (creating a connection in the SYN-RECEIVED state).
- Otherwise, silently drop the incoming SYN segment, thus leaving the previous incarnation of the connection in the TIME-WAIT state.
- If the previous incarnation of the connection did not use timestamps, then,
- If TCP timestamps would be enabled for the new incarnation of the connection, honour the incoming connection request.
- If TCP timestamps would not be enabled for the new incarnation of the connection, but the Sequence Number of the incoming SYN segment is larger than the last sequence number seen on the previous incarnation of the connection (for the same direction of the data transfer), then honour the incoming connection request (even if the sequence number of the incoming SYN segment falls within the receive window of the previous incarnation of the connection).

- Otherwise, silently drop the incoming SYN segment, thus leaving the previous incarnation of the connection in the TIME-WAIT state.

> *In the above explanation, the phrase "TCP timestamps would be enabled for the new incarnation for the connection" means that the incoming SYN segment contains a TCP Timestamps option (i.e., the client has enabled TCP timestamps), and that the SYN/ACK segment that would be sent in response to it would also contain a Timestamps option (i.e., the server has enabled TCP timestamps). In such a scenario, TCP timestamps would be enabled for the new incarnation of the connection.*
>
> *The "last sequence number seen on the previous incarnation of the connection (for the same direction of the data transfer)" refers to the last sequence number used by the previous incarnation of the connection (for the same direction of the data transfer), and not to the last value seen in the* ***`Sequence Number`*** *field of the corresponding segments. That is, it refers to the sequence number corresponding to the* ***`FIN`*** *flag of the previous incarnation of the connection, for that direction of the data transfer.*
>
> *The processing rules proposed in this Section do not comply with one of the requirements in the upcoming RFC "Improving TCP's Robustness to Blind In-Window Attacks" [Ramaiah et al, 2008], which requires implementations to send an ACK in response to in-window SYN segments received for connections in any of the synchronized states (including the TIME-WAIT state).*

Many implementations do not include the TCP timestamp option when performing the above heuristics, thus imposing stricter constraints on the generation of Initial Sequence Numbers, the average data transfer rate of the connections, and the amount of data transferred with them. RFC 793 [Postel, 1981c] states that the ISN generator should be incremented roughly once every four microseconds (i.e., roughly 250000 times per second). As a result, any connection that transfers more than 250000 bytes of data at more than 250 KB/s could lead to scenarios in which the last sequence number seen on a connection that moves into the TIME-WAIT state may still be greater than the sequence number of an incoming SYN segment that aims at creating a new incarnation of the same connection. In those scenarios, the 4.4BSD heuristics would fail, and therefore the connection request would usually time out. By including the TCP timestamp option in the heuristics described above, all these constraints are greatly relaxed.

It is clear that the use of TCP timestamps for the heuristics described above depends on the timestamps to be monotonically increasing across connections between the same two TCP endpoints. Therefore, we strongly advice to generate timestamps as described in Section 4.7.1.

11.1.3. Security/Compartment

Section 3.9 (page 71) of RFC 793 [Postel, 1981c] 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.

A discussion of the IP security options relevant to this section can be found in Section 3.13.2.12, Section 3.13.2.13, and Section 3.13.2.14 of [CPNI, 2008].

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.

It is interesting to note that for connections in the ESTABLISHED state, this check is performed after validating the TCP **`Sequence Number`** and checking the **`RST`** bit, but before validating the Acknowledgement field. Therefore, even if the stricter validation of the **`Acknowledgement`** field (described in Section 3.4) was implemented, it would not help to mitigate this attack vector.

This attack vector can be easily mitigated by relaxing the reaction to TCP segments with "incorrect" security/compartment values: if the security/compartment field does not match the value recorded in the corresponding TCB, TCP should not abort the connection, but simply discard the corresponding packet. Additionally, this whole event should be logged as a security violation.

11.1.4. Precedence

Section 3.9 (page 71) of RFC 793 [Postel, 1981c] states that if the IP **`Precedence`** of an incoming segment does not exactly match the Precedence recorded 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 IP **`Precedence`** that is different from that recorded in the corresponding TCB and, as a result, the attacked connection would be reset.

It is interesting to note that for connections in the ESTABLISHED state, this check is performed after validating the TCP **`Sequence Number`** and checking the **`RST`** bit, but before validating the **`Acknowledgement`** field. Therefore, even if the stricter validation of the **`Acknowledgement`** field (described in Section 3.4) were implemented, it would not help to mitigate this attack vector.

This attack vector can be easily mitigated by relaxing the reaction to TCP segments with "incorrect" IP **`Precedence`** values. That is, even if the Precedence field does not match the value recorded in the corresponding TCB, TCP should not abort the connection, and should instead continue processing the segment as specified by RFC 793.

It is interesting to note that resetting a connection due to a change in the Precedence value might have a negative impact on interoperability. For example, the packets that correspond to the connection could temporarily take a different internet path, in which some middle-box could re-mark the Precedence field (due to administration policies at the network to be transited). In such a scenario, an implementation following the advice in RFC 793 would abort the connection, when the connection would have probably survived.

While the IPv4 **`Type of Service`** field (and hence the **`Precedence`** field) has been redefined by the **`Differentiated Services`** (**`DS`**) field specified in RFC 2474 [Nichols et al, 1998], RFC 793 [Postel, 1981c] was never formally updated in this respect. We note that both legacy systems that have not been upgraded to implement the differentiated services architecture described in RFC 2475 [Blake et al, 1998] and current implementations that have extrapolated the discussion of the **`Precedence`** field to the **`Differentiated Services`** field may still be vulnerable to the connection reset vector discussed in this section.

11.1.5. Illegal options

Section 4.2.2.5 of RFC 1122 [Braden, 1989] discusses the processing of TCP options. It states that TCP must be able to receive a TCP option in any segment, and must ignore without error any option it does not implement. Additionally, 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. Therefore, as discussed in Section 3.10 of this document, we advise TCP implementations to silently drop those TCP segments that contain illegal option lengths.

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] propose to enforce a more strict check on the Acknowledgement Number of incoming segments than that specified in RFC 793 [Postel, 1981c].

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

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 characterising the behaviour 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 behaviour 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

The attacker sends a FIN (or any packet without the **SYN** or the **ACK** flags set) to an open port. RFC 793 [Postel, 1981c] leaves the reaction to such segments unspecified. As a result, some implementations silently drop the received segment, while others respond with a RST. We advice implementations to silently drop 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 should be applied.

12.1.2. Bogus flag test

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

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

[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. Here we provide advice on how the corresponding RST segments should be constructed.

If the **ACK** bit of an incoming TCP segment is off, a **Sequence Number** of zero should 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** should 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.

RFC 793 [Postel, 1981c] 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 analysing the RTO of the target system. However, this fingerprinting technique has at least the following drawbacks:

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

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

RFC 793 [Postel, 1981c] 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 [Postel, 1981c], 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 by 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. We recommend TCP/IP stacks to implement this alternative processing of TCP segments for connections in the LISTEN state.

14.4. Maimon scan

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 [Postel, 1981c] 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

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 [Postel, 1981c] 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

The Internet Control Message Protocol (ICMP) is used in the Internet Architecture mainly to perform a fault-isolation function, that is, the group of actions that hosts and routers take to determine that there is some network failure [Clark, 1982].

When a router detects a network problem while trying to forward an IP packet, it usually sends an ICMP error message to the source host, to raise awareness of the network problem taking place. In the same way, there are a number of scenarios in which a host may generate an ICMP error message if it finds a problem while processing an IP datagram. The received ICMP errors are handed to the corresponding transport-protocol instance, which will usually perform a fault recovery function.

Unfortunately, ICMP can be exploited to perform a variety of attacks against TCP (and other similar protocols), which include blind connection-reset, blind throughput-reduction, and blind performance-degrading attacks. All of these attacks can be performed even with the attacker being off-path, without the need to sniff the packets that correspond to the attacked TCP connection.

While the security implications of ICMP have been known in the research community for a long time, there is not yet an official proposal on how to deal with these vulnerabilities. However, as a result of the disclosure process carried out by the UK's National Infrastructure Security Co-ordination Centre (NISCC) (during 2004 and 2005) and the publication of an IETF Internet-Draft [Gont, 2008a], virtually all current TCP implementations now incorporate some countermeasures for these attacks.

The next sections provide a description of the use of ICMP to perform attacks against TCP, and describe the set of countermeasures that have become the "de facto" standard to mitigate the impact of these vulnerabilities.

15.1. Internet Control Message Protocol

The specification of the ICMP protocol is spread among a number of documents. This section provides a roadmap to the ICMP documents that are relevant to TCP.

15.1.1. Internet Control Message Protocol for IP version 4 (ICMP)

RFC 792 [Postel, 1981b] is the base specification of the Internet Control Message Protocol (ICMP) to be used with the Internet Protocol version 4 (IPv4). It defines, among other things, a number of error messages that can be used by end-systems and intermediate-systems to report network errors to the sending host. Additionally, it defines the ICMP Source Quench message (type 4, code 0), which is meant to provide a mechanism for flow control and congestion control.

RFC 1122 [Braden, 1989] classifies ICMP error messages into those that indicate “soft errors”, and those that indicate “hard errors”, thus roughly defining the semantics of them.

RFC 1191 [Mogul and Deering, 1990] defines the Path-MTU Discovery (PMTUD) mechanism, which makes use of ICMP error messages of type 3 (Destination Unreachable), code 4 (fragmentation needed and DF bit set) to allow hosts to determine the MTU of an arbitrary internet path.

Finally, Appendix D of RFC 4301 [Kent and Seo, 2005] provides information about which ICMP error messages are produced by hosts, routers, or both.

15.1.2. Internet Control Message Protocol for IP version 6 (ICMPv6)

RFC 4443 [Conta et al, 2006] specifies the Internet Control Message Protocol (ICMPv6) to be used with the Internet Protocol version 6 (IPv6) [Deering and Hinden, 1998].

RFC 4443 [Conta et al, 2006] defines the “Packet Too Big” (type 2, code 0) error message, that is analogous to the ICMP “fragmentation needed and DF bit set” (type 3, code 4) error message. RFC 1981 [McCann et al, 1996] defines the Path MTU Discovery mechanism for IP Version 6, that makes use of these messages to determine the MTU of an arbitrary internet path.

Appendix D of RFC 4301 [Kent and Seo, 2005] provides information about which ICMPv6 error messages are generated by hosts, routers, or both.

15.2. Handling of ICMP error messages

RFC 1122 [Braden, 1989] states that a TCP must act on an ICMP error message passed up from the IP layer, directing it to the connection that elicited the error.

In order to allow ICMP messages to be demultiplexed by the receiving host, part of the original packet that elicited the message is included in the payload of the ICMP error message. Thus, the receiving host can use that information to match the ICMP error to the transport protocol instance that elicited it.

Neither RFC 793 [Postel, 1981c] nor RFC 1122 [Braden, 1989] recommend any validation checks on the received ICMP messages. Thus, as long as the ICMP payload contains the information that identifies an existing communication instance, it will be handed to the corresponding transport-protocol instance, and the corresponding action will be performed.

Therefore, in the case of TCP, an attacker could send a forged ICMP message to the attacked host, and, as long as he is able to guess the four-tuple that identifies the communication instance to be attacked, he will be able to use ICMP to perform a variety of attacks.

As discussed in [Watson, 2004], there are a number of scenarios in which an attacker may know or be able to guess the four-tuple that identifies a TCP connection. If we assume the attacker knows the two systems involved in the TCP connection to be attacked, both the

client-side and the server-side IP addresses will be known. Furthermore, as most Internet services use the so-called “well-known” ports, only the client port number would need to be guessed. This means that an attacker would need to send, in principle, at most 65536 packets to perform any ICMP-based attack against TCP. However, as many systems choose the port numbers they use for outgoing connections from a subset of the whole port number space and do not randomise the ephemeral port numbers, in practice fewer packets are needed to perform any of these attacks.

15.3 Constraints in the possible solutions

For ICMPv4, RFC 792 [Postel, 1981b] states that the internet header plus the first 64 bits of the packet that elicited the ICMP message are to be included in the payload of the ICMP error message. Thus, it is assumed that all data needed to identify a transport protocol instance and process the ICMP error message is contained in the first 64 bits of the transport protocol header. RFC 1122 [Braden, 1989] allows implementations to optionally include more data from the original packet than those required by the original ICMP specification. Finally, RFC 1812 [Baker, 1995] recommends that ICMP error messages should contain as much of the original datagram as possible without the length of the ICMP datagram exceeding 576 bytes.

Thus, for ICMP messages generated by hosts, we can only expect to get the entire IPv4 header of the original packet, plus the first 64 bits of its payload. For TCP, this means that the only fields that will be included in the ICMP payload are: the Source Port, the Destination Port, and the 32-bit TCP Sequence Number. This clearly imposes a constraint on the possible validation checks that can be performed, as there is not much information available on which these checks could be performed.

These constraints mean, for example, that even if TCP were signing its segments by means of the TCP MD5 signature option specified in RFC 2385 [Heffernan, 1998], this mechanism could not be used as a counter-measure against ICMP-based attacks, because, as ICMP messages include only a piece of the TCP segment that elicited the error, the MD5 signature could not be recalculated. In the same way, even if the attacked peer was authenticating its packets at the IP layer [Kent and Seo, 2005], because only a part of the original IP packet would be available, the signature used for authentication could not be recalculated, and thus this mechanism could not be used as a counter-measure against ICMP-based attacks against TCP.

For the IPv6 case, RFC 4443 [Conta et al, 2006] specifies that the payload of ICMPv6 error messages includes as many octets from the IPv6 packet that elicited the ICMPv6 error message as will fit without making the resulting ICMPv6 packet exceed the minimum IPv6 MTU (1280 octets). Thus, more information is available than in the IPv4 case.

Hosts could require ICMP error messages to be authenticated (e.g., by means of IPsec), in order to act upon them. However, while this requirement could make sense for those ICMP error messages sent by hosts, it would not be feasible for those ICMP error messages generated by routers, as this would imply either that the attacked host should have a security association with every existing router, or that it should be able to establish one dynamically.

The current level of deployment of protocols for dynamic establishment of security associations makes this unfeasible. Also, in some cases, such as embedded devices, the processing power requirements of authentication might not allow IPsec authentication to be implemented effectively.

15.4. General countermeasures against ICMP attacks

There are a number of countermeasures that can be implemented to eliminate or mitigate the impact of ICMP-based attacks against TCP. The general countermeasures discussed in the following subsections help to mitigate many ICMP-based attacks against TCP. Rather than being alternative countermeasures, they can be implemented together to increase the protection against these attacks.

15.4.1. TCP sequence number checking

The current specifications do not impose any validity checks on the TCP segment that is contained in the ICMP payload. For instance, no checks are performed to verify that a received ICMP error message has been elicited by a segment that was “in flight” to destination. Thus, even stale ICMP error messages will be acted upon.

TCP should check that the TCP **`Sequence Number`** contained in the payload of the ICMP error message should be within the range of the data already sent but not yet acknowledged. That is,

$$SND.UNA =< \textbf{Sequence Number} < SND.NXT$$

If an ICMP error message does not pass this check, it should be silently dropped.

Even if an attacker were able to guess the four-tuple that identifies the TCP connection, this additional check would reduce the possibility of considering a forged ICMP packet as valid to *FlightSize*/2^{32} (where *FlightSize* is the number of data bytes already sent to the remote peer, but not yet acknowledged, as defined in RFC 2581 [Allman et al, 1999]). For connections in the SYN-SENT or SYN-RECEIVED states, this would reduce the possibility of considering a forged ICMP packet as valid to $1/2^{32}$. For a TCP endpoint with no data “in flight”, this would completely eliminate the possibility of success of these attacks.

This check has been incorporated by most major implementations of TCP.

It is important to note that while this check greatly increases the number of packets required to perform any of the attacks discussed in this document, this may not be enough in those scenarios in which bandwidth is easily available, and/or large TCP windows are in use (e.g., by means of the mechanism specified in RFC 1323 [Jacobson et al, 1992]). Therefore, implementation of the attack-specific countermeasures discussed in this document is strongly recommended.

15.4.2. Port randomisation

As discussed in the previous sections, in order to perform any of these ICMP-based attacks, an attacker would need to guess (or know) the four-tuple that identifies the connection to be attacked. Increasing the port number range used for outgoing TCP connections, and obfuscating the ephemeral port numbers used for outgoing TCP connections would make it harder for an attacker to perform any of these blind attacks against TCP.

Section 3.1 of this document discusses TCP ephemeral port randomisation in great detail.

15.4.3. Filtering ICMP error messages based on the ICMP payload

The source address of ICMP error messages does not need to be forged to perform the ICMP-based attacks against TCP. Therefore, simple filtering based on the source address of ICMP error messages does not serve as a counter-measure against these attacks. However, a more advanced packet filtering could be implemented in firewalls and other middle-boxes, which could help to mitigate these attacks. Firewalls implementing such advanced filtering would look at the payload of the ICMP error messages, and perform ingress and egress packet filtering based on the source IP address of the IP header contained in the payload of the ICMP error message.

[Gont, 2006] provides a discussion of filtering of ICMP messages based on the ICMP payload.

15.5. Blind connection-reset attack

15.5.1. Description

When TCP is handed an ICMP error message, it will perform its fault recovery function, as follows:

- If the network problem being reported is a hard error, TCP will abort the corresponding connection.
- If the network problem being reported is a soft error, TCP will just record this information, and repeatedly retransmit its data until they either get acknowledged, or the connection times out.

RFC 1122 [Braden, 1989] states that a host should abort a connection when receiving an ICMP error message that indicates a “hard error”, and states that ICMP error messages of type 3 (Destination Unreachable) codes 2 (protocol unreachable), 3 (port unreachable), and 4 (fragmentation needed and DF bit set) should be considered to indicate hard errors.

> *While RFC 4301 [Conta et al, 2006] did not exist when RFC 1122 was published, one could extrapolate the concept of “hard errors” to ICMPv6 error messages of type 1 (Destination unreachable) codes 1 (communication with destination administratively prohibited), and 4 (port unreachable).*

Thus, an attacker could use ICMP to perform a blind connection-reset attack. That is, even being off-path, an attacker could reset any TCP connection taking place by sending any ICMP

error message that indicates a “hard error”, to either of the two TCP endpoints of the connection. Because of TCP’s fault recovery policy, the connection would be immediately aborted.

As discussed in Section 15.2, all an attacker needs to know to perform such an attack is the socket pair that identifies the TCP connection to be attacked. In some scenarios, the IP addresses and port numbers in use may be easily guessed or known to the attacker [Watson, 2004].

Some stacks are known to propagate ICMP errors across TCP connections, increasing the impact of this attack, as a single ICMP packet could bring down all the TCP connections between the corresponding peers.

It is important to note that even if TCP itself were protected against the blind connection-reset attack described in [Watson, 2004] and [NISCC, 2004], by means of IPsec authentication [Kent and Seo, 2005], by means of the TCP MD5 signature option specified in RFC 2385 [Heffernan, 1998], or by means of the mechanism proposed in [Ramaiah et al, 2008], the blind connection-reset attack described in this document could still succeed.

15.5.2. Attack-specific countermeasures

Changing the reaction to hard errors

An analysis of the circumstances in which ICMP messages that indicate hard errors may be received can shed some light on how to eliminate the impact of ICMP-based blind connection-reset attacks.

ICMP type 3 (Destination Unreachable), code 2 (protocol unreachable)

This ICMP error message indicates that the host sending the ICMP error message received a packet meant for a transport protocol it does not support. For connection-oriented protocols such as TCP, one could expect to receive such an error as the result of a connection-establishment attempt. However, it would be strange to get such an error during the life of a connection, as this would indicate that support for that transport protocol has been removed from the host sending the error message during the life of the corresponding connection. Thus, it would be fair to treat ICMP protocol unreachable error messages as soft errors if they are meant for connections that are in synchronised states. For TCP, this means TCP should treat ICMP protocol unreachable error messages as soft errors if they are meant for connections that are in the ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK or TIME-WAIT states.

ICMP type 3 (Destination Unreachable), code 3 (port unreachable)

This error message indicates that the host sending the ICMP error message received a packet meant for a socket {IP address, port number} on which there is no process listening. Those transport protocols which have their own mechanisms for notifying this condition should not be receiving these error messages. However, RFC 1122 [Braden, 1989] states that even those transport protocols that have their own mechanism for notifying the sender that a port is unreachable must nevertheless accept an ICMP Port Unreachable for the same purpose. For security and robustness reasons, it would be fair to treat ICMP port unreachable messages as

soft errors when they are meant for protocols that have their own mechanism for reporting this error condition.

ICMP type 3 (Destination Unreachable), code 4 (fragmentation needed and DF bit set)
This error message indicates that an intermediate node needed to fragment a datagram, but the **DF** (Don't Fragment) bit in the IPv4 header was set. Those systems that do not implement the PMTUD mechanism should not be sending their IP packets with the **DF** bit set, and thus should not be receiving these ICMP error messages. Thus, it would be fair for them to treat this ICMP error message as indicating a soft error, therefore not aborting the corresponding connection when such an error message is received. On the other hand, and for obvious reasons, those systems implementing the Path-MTU Discovery (PMTUD) mechanism specified in RFC 1191 [Mogul and Deering, 1990] and RFC 1981 [McCann et al, 1996] should not abort a corresponding connection when such an ICMP error message is received.

ICMPv6 type 1 (Destination Unreachable), code 1 (communication with destination administratively prohibited)
This error message indicates that the destination is unreachable because of an administrative policy. For connection-oriented protocols such as TCP, one could expect to receive such an error as the result of a connection-establishment attempt. Receiving such an error for a connection in any of the synchronised states would mean that the administrative policy changed during the life of the connection. Therefore, while it would be possible for a firewall to be reconfigured during the life of a connection, it would be fair, for security and robustness reasons, to ignore these messages for connections that are in the ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK or TIME-WAIT states.

ICMPv6 type 1 (Destination Unreachable), code 4 (port unreachable)
This error message is analogous to the ICMP type 3 (Destination Unreachable), code 3 (Port unreachable) error message discussed above. Therefore, the same considerations apply.

Therefore, TCP should treat all ICMP error messages as indicating "soft errors" when they are meant for connections in any of the synchronised states and therefore should not abort the corresponding connection upon receipt of them. Also, as discussed in Section 15.5.1, hosts should not extrapolate ICMP errors across TCP connections.

In case the received message was legitimate, it would mean that the "hard error" condition appeared during the life of the connection. However, there is no reason to think that in the same way this error condition appeared, it would not get solved in the near term. Therefore, treating the received ICMP error messages as "soft errors" would make TCP more robust, and could avoid TCP from aborting a TCP connection unnecessarily. Aborting the connection would be to ignore the valuable feature of the Internet that for many internal failures it reconstructs its function without any disruption of the end points [Clark, 1982].

It is interesting to note that, as ICMP error messages are unreliable, transport protocols should not depend on them for correct functioning. In the event one of these messages was legitimate, the corresponding connection would eventually time out. Also, applications may still be notified asynchronously about the received error messages, and thus may still abort their connections on their own if they consider it appropriate.

This counter-measure has become the “de facto” standard for dealing with the so-called ICMP “hard errors” when they are received for connection in any of the synchronised states.

Delaying the connection reset

An alternative counter-measure would be, in the case of connections in any of the synchronised states, to honour the ICMP error messages only if there is no progress on the connection. Rather than immediately aborting a connection, a TCP would abort a connection only after an ICMP error message indicating a hard error has been received, and the corresponding data have already been retransmitted more than some specified number of times.

The rationale behind this proposed fix is that if a host can make forward progress on a connection, it can completely disregard the "hard errors" being indicated by the received ICMP error messages. However, while this counter-measure could be useful, the one described earlier in this section is easier to implement, and provides increased protection against this type of attack.

15.6. Blind throughput-reduction attack

15.6.1. Description

RFC 1122 [Braden, 1989] states that hosts must react to ICMP Source Quench messages by slowing transmission on the connection. Thus, an attacker could send ICMP Source Quench (type 4, code 0) messages to a TCP endpoint to make it reduce the rate at which it sends data to the other end-point of the connection. RFC 1122 further adds that the recommended procedure is to put the corresponding connection in the slow-start phase of the TCP’s congestion control algorithm (described at the time in [Jacobson, 1988], and currently standardised by RFC 2581 [Allman et al, 1999]). In the case of those implementations that use an initial congestion window of one segment, a sustained attack would reduce the throughput of the attacked connection to about *SMSS* (*Sender Maximum Segment Size*) bytes per RTT (round-trip time). The throughput achieved during attack might be higher if a larger initial congestion window is in use, as specified in RFC 3390 [Allman et al, 2002].

15.6.2. Attack-specific countermeasures

RFC 1122 [Braden, 1989] states that hosts must react to ICMP Source Quench messages by slowing transmission on the connection. However, as discussed in RFC 1812 [Baker, 1995], research seems to suggest ICMP Source Quench is an ineffective (and unfair) antidote for congestion. RFC 1812 further states that routers should not send ICMP Source Quench messages in response to congestion. On the other hand, TCP implements its own congestion control mechanisms [Allman et al, 1999] [Ramakrishnan et al, 2001], that do not depend on ICMP Source Quench messages. Thus, hosts should silently drop ICMP Source Quench messages that are meant for TCP connections.

15.7. Blind performance-degrading attack

15.7.1. Description

When one IP host has a large amount of data to send to another host, the data will be transmitted as a series of IP datagrams. It is usually preferable that these datagrams be of the largest size that does not require fragmentation anywhere along the path from the source to the destination. This datagram size is referred to as the Path MTU (PMTU), and is equal to the minimum of the MTUs of each hop in the path [Mogul and Deering, 1990].

A technique called “Path MTU Discovery” (PMTUD) mechanism lets IP hosts determine the Path MTU of an arbitrary internet path. RFC 1191 [Mogul and Deering, 1990] and RFC 1981 [McCann et al, 1996] specify the PMTUD mechanism for IPv4 and IPv6, respectively.

The PMTUD mechanism for IPv4 uses the Don’t Fragment (**DF**) bit in the IPv4 header to dynamically discover the Path MTU. The basic idea behind the PMTUD mechanism is that a source host assumes that the MTU of the path is that of the first hop, and sends all its datagrams with the **DF** bit set. If any of the datagram is too large to be forwarded without fragmentation by some intermediate router, the router will discard the corresponding datagram, and will return an ICMP “Destination Unreachable” (type 3) “fragmentation needed and DF set” (code 4) error message to sending host. This message will report the MTU of the constricting hop, so that the sending host can reduce the assumed Path-MTU accordingly.

For IPv6, intermediate systems do not fragment packets. Thus, there is an “implicit” DF bit set in every packet sent on an IPv6 network. If any of the datagrams is too large to be forwarded without fragmentation by some intermediate router, the router will discard the corresponding datagram, and will return an ICMPv6 “Packet Too Big” (type 2, code 0) error message to the sending host. This message will report the MTU of the constricting hop, so that the sending host can reduce the assumed Path-MTU accordingly.

As discussed in both RFC 1191 [Mogul and Deering, 1990] and RFC 1981 [McCann et al, 1996], the Path-MTU Discovery mechanism can be used to attack TCP. An attacker could send a forged ICMP “Destination Unreachable, fragmentation needed and DF set” error message (or its ICMPv6 counterpart) to the sending host, advertising a small Next-Hop MTU. As a result, the attacked system would reduce the size of the packets it sends for the corresponding connection accordingly.

The effect of this attack is two-fold. On one hand, it will increase the headers/data ratio, thus increasing the overhead needed to send data to the remote TCP end-point. On the other hand, if the attacked system wanted to keep the same throughput it was achieving before being attacked, it would have to increase the packet rate. On virtually all systems this will lead to an increase in the IRQ (Interrrupt ReQuest) rate, thus increasing processor utilisation, and degrading the overall system performance. A particular scenario that may take place is that in which an attacker reports a Next-Hop MTU smaller than or equal to the amount of bytes needed for headers (IP header, plus TCP header). For example, if the attacker reports a Next-Hop MTU of 68 bytes, and the amount of bytes used for headers (IPv4 header, plus TCP header) is larger than 68 bytes, the assumed Path-MTU will not even allow the attacked host

to send a single byte of application data without fragmentation. This particular scenario might lead to unpredictable results. Another possible scenario is that in which a TCP connection is being secured by means of IPsec [Kent and Seo, 2006]. If the Next-Hop MTU reported by the attacker is smaller than the amount of bytes needed for headers (IP and IPsec, in this case), the assumed Path-MTU will not even allow the attacked host to send a single byte of the TCP header without fragmentation. This is another scenario that might lead to unpredictable results.

For IPv4, the reported Next-Hop MTU could be as low as 68 octets, as RFC 791 [Postel, 1981a] requires every internet module to be able to forward a datagram of 68 octets without further fragmentation. For IPv6, the reported Next-Hop MTU could be as low as 1280 octets (the minimum IPv6 MTU, as specified by RFC 2460 [Deering and Hinden, 1998]).

Recently, the PMTUD WG [PMTUDWG, 2007] of the IETF produced the document RFC 4821 [Mathis and Heffner, 2007], which specifies a mechanism for discovering the Path-MTU known as “Packetization Layer Path MTU Discovery” (PLPMTUD), which does not rely on ICMP error messages. This mechanism can be implemented as a replacement for the traditional Path-MTU Discovery mechanism specified in RFC 1191 [Mogul and Deering, 1990] and RFC 1981 [McCann et al, 1996], or only for black-hole detection.

> *“Black-holes” are caused by routers that discard packets that are too large to be forwarded without fragmentation (and have the IP* **DF** *bit set), without sending an ICMP error message to the sending endpoint. An equivalent scenario is that in which the router that discards the packets does send an ICMP error message to the sending endpoint, but some intermediate system (such as a firewall) consistently drops the corresponding ICMP error messages [Lahey, 2000].*

While replacement of the traditional Path-MTU Discovery mechanism with PLPMTUD would eliminate the attack vector described in this section, the convergence time of PLPMTUD is typically longer than that of the traditional PMTUD mechanism, and thus a number TCP implementers seem to be unwilling to implement PLPMTUD as a complete replacement for the traditional PMTUD mechanism.

15.7.2. Attack-specific countermeasures

Henceforth, we will refer to both ICMP “fragmentation needed and DF bit set” and ICMPv6 “Packet Too Big” error messages as “ICMP Packet Too Big” error messages.

In addition to the general validation check described in Section 15.4.1, processing of ICMP “Packet Too Big” error message could be delayed as described in Section 15.5.2, to greatly mitigate the impact of this attack.

This would mean that upon receipt of an ICMP “Packet Too Big” error message, TCP would just record this information, and would honour it only when the corresponding data had already been retransmitted a specified number of times.

While this policy would mitigate the impact of the attack against the PMTUD mechanism, it would also mean that it might take TCP more time to discover the Path-MTU for a TCP

connection. This would be particularly annoying for connections that have just been established, as it might take TCP several transmission attempts (and the corresponding timeouts) before it discovers the PMTU for the corresponding connection. Thus, this policy would increase the time it takes for data to begin to be received at the destination host.

We would like to protect TCP from the attack against the PMTUD mechanism, while still allowing TCP to quickly determine the initial Path-MTU for a connection. To achieve both goals, we can divide the traditional PMTUD mechanism into two stages: Initial Path-MTU Discovery and Path-MTU Update.

The Initial Path-MTU Discovery stage is when TCP tries to send segments that are larger than the ones that have so far been sent and acknowledged for this connection. That is, in the Initial Path-MTU Discovery stage TCP has no record of these large segments getting to the destination host, and thus it would be fair to believe the network when it reports that these packets are too large to reach the destination host without being fragmented.

The Path-MTU Update stage is when TCP is asked to reduce the size of the segments it sends to a value that is equal to or smaller than that of the largest TCP segment that has so far been sent and acknowledged for this connection. During the Path-MTU Update stage, TCP already has knowledge of the estimated Path-MTU for the given connection. Thus, it would be fair to be more cautious with the errors being reported by the network.

In order to allow TCP to distinguish segments between those performing Initial Path-MTU Discovery and those performing Path-MTU Update, two new variables would need to be introduced to TCP: *maxsizeacked* and *maxsizesent*.

maxsizesent would hold the size (in octets) of the largest packet that has so far been sent for this connection. It would be initialized to 68 (the minimum IPv4 MTU) when the underlying internet protocol is IPv4, and would be initialized to 1280 (the minimum IPv6 MTU) when the underlying internet protocol is IPv6. Whenever a packet larger than *maxsizesent* octets is sent, *maxsizesent* should be set to that value.

On the other hand, *maxsizeacked* would hold the size (in octets) of the largest packet that has so far been acknowledged for this connection. It would be initialized to 68 (the minimum IPv4 MTU) when the underlying internet protocol is IPv4, and would be initialized to 1280 (the minimum IPv6 MTU) when the underlying internet protocol is IPv6. Whenever an acknowledgement for a packet larger than *maxsizeacked* octets is received, *maxsizeacked* should be set to the size of that acknowledged packet.

Upon receipt of an ICMP “Packet Too Big” error message, the Next-Hop MTU claimed by the ICMP message (henceforth “*claimedmtu*”) should be compared with *maxsizesent*. If *claimedmtu* is equal to or larger than *maxsizesent*, then the ICMP error message should be silently discarded. The rationale for this policy is that the ICMP error message cannot be legitimate if it claims to have been elicited by a packet larger than the largest packet we have so far sent for this connection.

If this check is passed, *claimedmtu* should be compared with *maxsizeacked*. If *claimedmtu* is equal to or larger than *maxsizeacked*, TCP is supposed to be in the Initial Path-MTU Discovery stage, and thus the ICMP “Packet Too Big” error message should be honoured immediately. That is, the assumed Path-MTU should be updated according to the Next-Hop MTU claimed in the ICMP error message. Also, *maxsizesent* should be reset to the minimum MTU of the internet protocol in use (68 for IPv4, and 1280 for IPv6).

On the other hand, if *claimedmtu* is smaller than *maxsizeacked*, TCP is supposed to be in the Path-MTU Update stage. At this stage, TCP should be more cautious with the errors being reported by the network, and should therefore just record the received error message, and delay the update of the assumed Path-MTU.

To perform this delay, one new variable and one new parameter should be introduced to TCP: *nsegrto* and *MAXSEGRTO*. *nsegrto* will hold the number of times a specified segment has timed out. It should be initialized to zero, and should be incremented by one every time the corresponding segment times out. *MAXSEGRTO* would specify the number of times a given segment must timeout before an ICMP “Packet Too Big” error message can be honoured, and could be set, in principle, to any value greater than or equal to 0.

Thus, if *nsegrto* is greater than or equal to *MAXSEGRTO*, and there’s a pending ICMP “Packet Too Big” error message, the corresponding error message should be honoured. *maxsizeacked* should be set to *claimedmtu*, and *maxsizesent* should be set to 68 (for IPv4) or 1280 (for IPv6).

If while there is a pending ICMP “Packet Too Big” error message the TCP **`Sequence Number`** claimed by the pending ICMP error message is acknowledged (i.e., an ACK that acknowledges that sequence number is received), then the “pending error” condition should be cleared.

The rationale behind performing this delayed processing of ICMP “Packet Too Big” error messages is that if there is progress on the connection, the ICMP “Packet Too Big” errors must be a false claim. By checking for progress on the connection, rather than just for staleness (i.e., checking the embedded TCP **`Sequence Number`**) of the received ICMP messages, TCP is protected from attack even if the offending ICMP messages are “in window”, and therefore as a corollary, is made more robust to spurious ICMP messages elicited by, for example, corrupted TCP segments.

MAXSEGRTO can be set, in principle, to any value greater than or equal to 0. Setting *MAXSEGRTO* to 0 would make TCP perform the traditional PMTUD mechanism defined in RFC 1191 [Mogul and Deering, 1990] and RFC 1981 [McCann et al, 1996]. A *MAXSEGRTO* of 1 should provide enough protection for most scenarios. In any case, implementations are free to choose higher values for this constant. *MAXSEGRTO* could be a function of the Next-Hop MTU claimed in the received ICMP “Packet Too Big” message. That is, higher values for *MAXSEGRTO* could be imposed when the received ICMP “Packet Too Big” message claims a Next-Hop MTU that is smaller than some specified value.

In the event a higher level of protection was desired at the expense of a higher delay in the discovery of the Path-MTU, an implementation could consider TCP to always be in the Path-MTU Update stage, thus always delaying the update of the assumed Path-MTU.

The current PMTUD mechanism, as specified by RFC 1191 [Mogul and Deering, 1990] and RFC 1981 [McCann et al, 1996], still suffers from some functionality problems described in RFC 2923 [Lahey, 2000] that the proposed countermeasure does not aim to address. A mechanism that addresses those issues is specified in RFC 4821 [Mathis and Heffner, 2007].

15.7.3. The countermeasure for the PMTUD attack in action

This section shows the proposed counter-measure for the ICMP attack against the PMTUD mechanism in action. It shows both how the counter-measure protects PMTUD from being exploited and how the counter-measure works in normal scenarios. This section assumes the PMTUD-specific counter-measure is implemented in addition to the TCP sequence number check proposed in Section 15.4.1.

Figure 19 illustrates a hypothetical scenario in which two hosts are connected by means of three intermediate routers. It also shows the MTU of each hypothetical hop. All the following subsections assume the network setup of this figure.

Also, for simplicity's sake, all subsections assume an IPv4 header of 20 octets and a TCP header of 20 octets. Thus, for example, when the PMTU is assumed to be 1500 octets, TCP will send segments that contain at most 1460 octets of data.

Finally, all the following subsections assume the TCP implementation at Host 1 has chosen an *MAXSEGRTO* of 1.

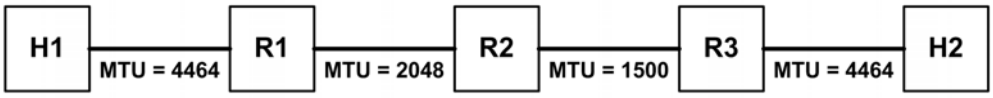

Figure 19: Hypothetical scenario

Normal operation for bulk transfers

This subsection shows the proposed counter-measure in normal operation, when a TCP connection is used for bulk transfers. That is, it shows how the proposed counter-measure works when there is no attack taking place, and a TCP connection is used for transferring large amounts of data. It assumes that just after the connection is established, one of the TCP endpoints begins to transfer data in packets that are as large as possible.

```
   Host 1                                          Host 2

1.    -->           <SEQ=100><CTL=SYN>            -->
2.    <--     <SEQ=X><ACK=101><CTL=SYN,ACK>       <--
3.    -->      <SEQ=101><ACK=X+1><CTL=ACK>        -->
4.    --> <SEQ=101><ACK=X+1><CTL=ACK><DATA=4424>  -->
5.        <--- ICMP "Packet Too Big" MTU=2048, TCPseq#=101 <--- R1
6.    --> <SEQ=101><ACK=X+1><CTL=ACK><DATA=2008>  -->
```

Figure 20: Normal operation for bulk transfers

nsegrto is initialised to zero. Both *maxsizeacked* and *maxsizesent* are initialized to the minimum MTU for the internet protocol being used (68 for IPv4, and 1280 for IPv6).

In lines 1 to 3 the three-way handshake takes place, and the connection is established. In line 4, H1 tries to send a full-sized TCP segment. As described by RFC 1191 and RFC 1981, in this case TCP will try to send a segment with 4424 bytes of data, which will result in an IP packet of 4464 octets. Therefore, *maxsizesent* is set to 4464. When the packet reaches R1, it elicits an ICMP "Packet Too Big" error message.

In line 5, H1 receives the ICMP error message, which reports a Next-Hop MTU of 2048 octets. After performing the TCP sequence number check described in Section 15.4.1, the Next-Hop MTU reported by the ICMP error message (*claimedmtu*) is compared with *maxsizesent*. As it is smaller than *maxsizesent*, it passes the check, and thus is then compared with *maxsizeacked*. As *claimedmtu* is larger than *maxsizeacked*, TCP assumes that the corresponding TCP segment was performing the Initial PMTU Discovery. Therefore, the TCP at H1 honours the ICMP message by updating the assumed Path-MTU. *maxsizesent* is reset to the minimum MTU of the internet protocol in use (68 for IPv4, and 1280 for IPv6).

In line 6, the TCP at H1 sends a segment with 2008 bytes of data, which results in an IP packet of 2048 octets. *maxsizesent* is thus set to 2008 bytes. When the packet reaches R2, it elicits an ICMP “Packet Too Big” error message.

In line 7, H1 receives the ICMP error message, which reports a Next-Hop MTU of 1500 octets. After performing the TCP sequence number check, the Next-Hop MTU reported by the ICMP error message (*claimedmtu*) is compared with *maxsizesent*. As it is smaller than *maxsizesent*, it passes the check, and thus is then compared with *maxsizeacked*. As *claimedmtu* is larger than *maxsizeacked*, TCP assumes that the corresponding TCP segment was performing the Initial Path-MTU Discovery. Therefore, the TCP at H1 honours the ICMP message by updating the assumed Path-MTU. *maxsizesent* is reset to the minimum MTU of the internet protocol in use.

In line 8, the TCP at H1 sends a segment with 1460 bytes of data, which results in an IP packet of 1500 octets. *maxsizesent* is thus set to 1500. This packet reaches H2, where it elicits an acknowledgement (ACK) segment.

In line 9, H1 finally gets the acknowledgement for the data segment. As the corresponding packet was larger than *maxsizeacked*, TCP updates *maxsizeacked*, setting it to 1500. At this point TCP has discovered the Path-MTU for this TCP connection.

Operation during Path-MTU changes

Let us suppose a TCP connection between H1 and H2 has already been established, and that the Path-MTU for the connection has already been discovered to be 1500. At this point, both *maxsizesent* and *maxsizeacked* are equal to 1500, and *nsegrto* is equal to 0. Suppose some time later the Path-MTU decreases to 1492. For simplicity, let us suppose that the Path-MTU has decreased because the MTU of the link between R2 and R3 has decreased from 1500 to 1492. Figure 21 illustrates how the proposed counter-measure would work in this scenario.

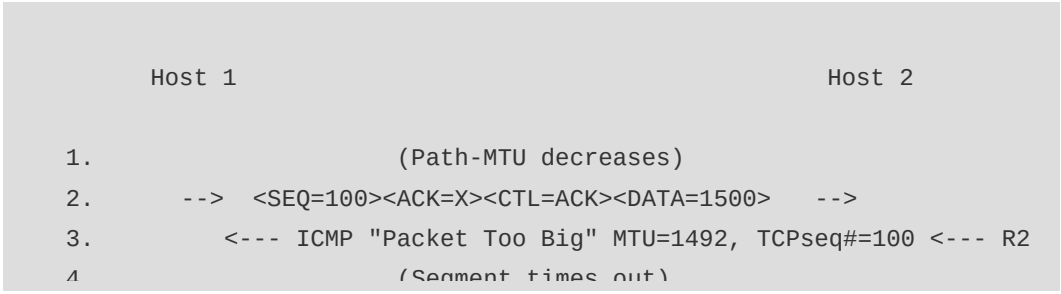

Figure 21: Operation during Path-MTU changes

In line 1, the Path-MTU for this connection decreases from 1500 to 1492. In line 2, the TCP at H1, without being aware of the Path-MTU change, sends a 1500-byte packet to H2. When the packet reaches R2, it elicits an ICMP “Packet Too Big” error message.

In line 3, H1 receives the ICMP error message, which reports a Next-Hop MTU of 1492 octets. After performing the TCP sequence number check, the Next-Hop MTU reported by the ICMP error message (*claimedmtu*) is compared with *maxsizesent*. As *claimedmtu* is smaller than *maxsizesent*, it is then compared with *maxsizeacked*. As *claimedmtu* is smaller than *maxsizeacked* (full-sized packets were getting to the remote end-point), this packet is assumed to be performing Path-MTU Update. And a “pending error” condition is recorded.

In line 4, the segment times out. Thus, *nsegrto* is incremented by 1. As *nsegrto* is greater than or equal to *MAXSEGRTO*, the assumed Path-MTU is updated. *nsegrto* is reset to 0, *maxsizeacked* is set to *claimedmtu*, and *maxsizesent* is set to the minimum MTU of the internet protocol in use.

In line 5, H1 retransmits the data using the updated Path-MTU, and thus *maxsizesent* is set to 1492. The resulting packet reaches H2, where it elicits an acknowledgement (ACK) segment.

In line 6, H1 finally gets the acknowledgement for the data segment. At this point TCP has discovered the new Path-MTU for this TCP connection.

Idle connection being attacked

Let us suppose a TCP connection between H1 and H2 has already been established, and the PMTU for the connection has already been discovered to be 1500. Figure 22 shows a sample time-line diagram that illustrates an idle connection being attacked.

```
Host 1                                          Host 2

1.     -->     <SEQ=100><ACK=X><CTL=ACK><DATA=50>      -->
2.     <--          <SEQ=X><ACK=150><CTL=ACK>          <--
3.          <--- ICMP "Packet Too Big" MTU=68, TCPseq#=100 <---
4.          <--- ICMP "Packet Too Big" MTU=68, TCPseq#=100 <---
5.          <--- ICMP "Packet Too Big" MTU=68, TCPseq#=100 <---
```

In line 1, H1 sends its last bunch of data. In line 2, H2 acknowledges the receipt of these data. Then the connection becomes idle. In lines 3, 4, and 5, an attacker sends forged ICMP “Packet Too Big” error messages to H1. Regardless of how many packets it sends and the TCP sequence number each ICMP packet includes, none of these ICMP error messages will pass the TCP sequence number check described in Section 15.4.1, as H1 has no unacknowledged data in flight to H2. Therefore, the attack does not succeed.

Active connection being attacked after discovery of the Path-MTU

Let us suppose an attacker attacks a TCP connection for which the PMTU has already been discovered. In this case, as illustrated in Figure 23, the PMTU would be found to be 1500 bytes. Figure 23 shows a possible packet exchange.

```
     Host 1                                          Host 2

1.    -->    <SEQ=100><ACK=X><CTL=ACK><DATA=1460>    -->
2.    -->    <SEQ=1560><ACK=X><CTL=ACK><DATA=1460>   -->
3.    -->    <SEQ=3020><ACK=X><CTL=ACK><DATA=1460>   -->
4.    -->    <SEQ=4480><ACK=X><CTL=ACK><DATA=1460>   -->
5.        <--- ICMP "Packet Too Big" MTU=68, TCPseq#=100 <---
6.    <--        <SEQ=X><CTL=ACK><ACK=1560>          <--
```

Figure 23: Active connection being attacked after discovery of PMTU

As we assume the Path-MTU has already been discovered, we assume both *maxsizesent* and *maxsizeacked* are equal to 1500. We also assume *nsegrto* is equal to zero, as there have been no segment timeouts.

In lines 1, 2, 3, and 4, H1 sends four data segments to H2. In line 5, an attacker sends a forged ICMP error message to H1. We assume the attacker is lucky enough to guess both the four-tuple that identifies the connection and a valid TCP sequence number. As the Next-Hop MTU claimed in the ICMP “Packet Too Big” message (*claimedmtu*) is smaller than

maxsizeacked, this packet is assumed to be performing Path-MTU Update. Thus, the error message is just recorded.

In line 6, H1 receives an acknowledgement for the segment sent in line 1, before it times out. At this point, the “pending error” condition is cleared, and the recorded ICMP “Packet Too Big” error message is discarded. Therefore, the attack does not succeed.

TCP peer attacked when sending small packets just after the three-way handshake

This subsection analyzes a corner-case in which a TCP peer that is sending small segments just after the connection has been established is attacked. The connection could be being used by protocols such as SMTP [Klensin, 2008] or HTTP [Fielding et al, 1999], for example, which usually behave like this.

Figure 24 shows a possible packet exchange for such scenario.

```
    Host 1                                          Host 2

1.   -->          <SEQ=100><CTL=SYN>             -->
2.   <--     <SEQ=X><ACK=101><CTL=SYN,ACK>       <--
3.   -->      <SEQ=101><ACK=X+1><CTL=ACK>        -->
4.   -->  <SEQ=101><ACK=X+1><CTL=ACK><DATA=100>  -->
5.   <--      <SEQ=X+1><ACK=201><CTL=ACK>        <--
6.   -->  <SEQ=201><ACK=X+1><CTL=ACK><DATA=100>  -->
7.   -->  <SEQ=301><ACK=X+1><CTL=ACK><DATA=100>  -->
8.        <--- ICMP "Packet Too Big" MTU=150, TCPseq#=101 <---
```

Figure 24: TCP peer attacked when sending small packets just after the three-way handshake

nsegrto is initialized to zero. Both *maxsizesent* and *maxsizeacked* are initialized to the minimum MTU for the internet protocol being used (68 for IPv4, and 1280 for IPv6).

In lines 1 to 3 the three-way handshake takes place, and the connection is established. At this point, the assumed Path-MTU for this connection is 4464. In line 4, H1 sends a small segment (which results in a 140-byte packet) to H2. *maxsizesent* is thus set to 140. In line 5 this segment is acknowledged, and thus *maxsizeacked* is set to 140.

In lines 6 and 7, H1 sends two small segments to H2. In line 8, while the segments from lines 6 and 7 are still in flight to H2, an attacker sends a forged ICMP “Packet Too Big” error message to H1. Assuming the attacker is lucky enough to guess a valid TCP sequence number, this ICMP message will pass the TCP sequence number check. The Next-Hop MTU reported by the ICMP error message (*claimedmtu*) will then be compared with *maxsizesent*. As *claimedmtu* will be larger than *maxsizesent*, the ICMP error message will be silently discarded. Therefore, the attack will not succeed.

15.7.4. Pseudo-code for the counter-measure for the blind performance-degrading attack

This section contains a pseudo-code version of the counter-measure to the blind performance-degrading attack described in the previous section. It is meant to provide guidance for developers on how to implement this counter-measure.

The pseudo-code makes use of the following variables, constants, and functions:

`ack`
Variable holding the acknowledgement number contained in the TCP segment that has just been received.

`acked_packet_size`
Variable holding the packet size (data, plus headers) the ACK that has just been received is acknowledging.

`adjust_mtu()`
Function that adjusts the MTU for this connection, according to the value of the variable “current_mtu”.

`claimedmtu`
Variable holding the Next-Hop MTU advertised by the ICMP “Packet Too Big” error message that has just been received.

`claimedtcpseq`
Variable holding the TCP sequence number contained in the payload of the ICMP "Packet Too Big" message that has just been received.

`current_mtu`
Variable holding the assumed Path-MTU for this connection.

`drop_message()`
Function that performs the necessary actions to drop the ICMP “Packet Too Big” error message being processed.

`initial_mtu`
Variable holding the MTU for new connections, as explained in RFC 1191 [Mogul and Deering, 1990] and RFC 1981 [McCann et al, 1996].

`maxsizeacked`
Variable holding the largest packet size (data plus headers) that has so for been acknowledged for this connection.

`maxsizesent`
Variable holding the largest packet size (data, plus headers) that has so for been sent for this connection.

`nsegrto`
Variable holding the number of times this segment has timed out.

`packet_size`
Variable holding the size of the IP datagram being sent

`pending_message`
Variable (flag) that indicates whether there is a pending ICMP “Packet Too Big” message to be processed.

`saved_tcpseq`
Variable that holds the TCP sequence number contained in the payload of an ICMP “Packet Too Big” error message whose processing was pending.

`saved_mtu`
Variable holding the Next-Hop MTU advertised by an ICMP “Packet Too Big” error message whose processing was pending.

`MINIMUM_MTU`
Constant holding the minimum MTU for the internet protocol in use (68 for IPv4, and 1280 for IPv6.

`MAXSEGRTO`
Constant holding the number of times a given segment must timeout before an ICMP “Packet Too Big” error message can be honoured.

EVENT: New TCP connection

```
current_mtu = initial_mtu;
maxsizesent = MINIMUM_MTU;
maxsizeacked = MINIMUM_MTU;
nsegrto = 0;
pending_message = 0;
```

EVENT: Segment is sent

```
if (packet_size > maxsizesent)
    maxsizesent = packet_size;
```

EVENT: Segment is received

```
if (acked_packet_size > maxsizeacked)
    maxsizeacked = acked_packet_size;

if (pending_message)
    if (ack > saved_tcpseq){
        pending_message = 0;
        nsegrto = 0;
    }
```

EVENT: ICMP "Packet Too Big" message is received

```
if (claimedtcpseq < SND.UNA || claimedtcpseq >= SND.NXT){
    drop_message();
}

else {
    if (claimedmtu >= maxsizesent || claimedmtu >= current_mtu)
        drop_message();

    else {
        if (claimedmtu > maxsizeacked){
            current_mtu = claimedmtu;
            maxsizesent = MINIMUM_MTU;
            adjust_mtu();
            drop_message();
        }

        else {
            pending_message = 1;
            saved_tcpseq = claimedtcpseq;
            saved_mtu = claimedmtu;
            drop_message();
        }
    }
}
```

EVENT: Segment times out

```
    nsegrto++;

    if (pending_message && nsegrto >= MAXSEGRTO){
        nsegrto = 0;
        pending_message = 0;
        maxsizeacked = saved_mtu;
        maxsizesent = MINIMUM_MTU;
        current_mtu = saved_mtu;
        adjust_mtu();
    }
```

Notes:

All comparisons between sequence numbers must be performed using sequence number arithmetic.

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 randomisation of the IP Identification field as described in Section 3.5.2 of [CPNI, 2008]. While randomisation 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. References

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

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, April 2002. 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, 2008c. *Generation of timestamps.*

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.

Hönes, 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.

Larsen, M., Gont, F. 2008. *Port Randomization*. IETF Internet-Draft (draft-ietf-tsvwg-port-randomization-02), work in progress.

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 USENET´s 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

Zúquete, 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.