TCP RESET GUARD – Protecting TCP Sessions from RST Attacks

John Egan **Melissa Hoang** **Ricky Chung**

Abstract:

We propose a new TCP option for preventing RST/FIN attacks on established TCP connections. RST/FIN attack prevention is crucial due to their ability to compromise the underlying internet routing infrastructure of BGP routers. Our proposed option improves upon previous mechanisms due to the fact it is backwards compatible, requires no changes to the basic functioning of the TCP protocol, and only incurs overhead on connection setup and teardown.

1.0 Introduction

TCP reset and FIN denials of service attacks have been around for more than a decade now, but there is still no elegant solution to the problem. The TCP RST/FIN attack works by sending a spoofed RST or FIN packet to the victim to abruptly close an established TCP connection. Imagine three hosts A, B, and C with an established connection between host A and B. To mount a TCP reset attack, host C will spoof a packet to B with A's IP address, port number and sequence number. C will set the RST or FIN flag in the packet so that when B gets this spoof packet it will close its connection to A believing that A has requested for the connection to be closed.

It was originally thought that this attack was hard because host C would need to guess the sequence number that A is using to communicate with B. A brute force attack on the sequence number would require trying every number from 1 up to the maximum of 2^{32} which is a big enough attack to be easily detected and is also not feasible for most attackers to pull off. Paul Watson on the other hand discovered that due to trends of increasing TCP window sizes and TCP acceptance of any packet in the window, a TCP RST/FIN attack needs to only try one sequence numbers for each window interval [Watson]. If the window size is 16K, an attacker needs to find the right window which takes 2^{32}/16K = 262,143 tries. Even worse TCP provides a field to increase the window size to up to 64K to increase performance. If a 64K window size is used, it will only take an attacker 65,537 tries before successfully mounting the attack. Watson provides experiments that show that on average for a DSL connection a TCP reset attack takes on average 3 minutes to succeed while on a T1 line it only takes an average 8 secs. Of course finding the right sequence number is not the only information needed for a TCP RST/FIN attack. The IP address and port numbers for both the source and destination need to be known. The destination address and port numbers are available online, since the destination host is usually a server, while the IP address of the source is easy to find. The source port can be difficult to guess right if they are allocated randomly but unfortunately due to overheads and difficulties, many operating systems allocate port numbers in a predictable manner which makes them easy for attackers to guess.

The difference between a RST and a FIN attack is that resets are handled right away. If a packet is sent with the RST flag set, it only needs to be in the right window for the connection to be closed. In a FIN attack, the packet needs to have the FIN flag set and also needs to be in the right window. The connection will not closed however until the sequence number that the FIN flag used is reached. This difference makes TCP RST attacks more dangerous.

The ability to prematurely close a connection will not bother most applications on the Internet but is extremely harmful to BGP. The BGP protocol relies on persistent TCP connections being maintained between its peers. One problem is that most BGP routers, if they lose their connection to a peer, will stop routing packets to the peer and will withdraw all route announcements for routes that passed through that peer. Furthermore if an attacker is able to shutdown BGP sessions, even if the router doesn't stop forwarding to the disconnected peers, the router will stop receiving updates from those peers it has been disconnected from which can cause it's routing tables to become out of date

We provide a simple backwards compatible transport layer solution to this attack that involves using the TCP option header to validate RST/FIN requests. Our proposal is to have each end calculate a connection unique secret value as well as a connection unique hash value that is derived from the secret value. The two hosts share the hash values during the TCP connection setup and then if either end wants to close the connection they send the secret value along with the RST or FIN packet so the other end can use the value to calculate the hash and verify it matches the hash they were sent during connection setup. This process runs into overhead in the connection setup and connection teardown but does not incur any per packet calculation overhead for an established connection like TCP MD5 does. Furthermore the overhead for connection setup and teardown is minimal compared to the amount of time it takes to establish or teardown a TCP connection.

The rest of this paper is structured as follow: Section 2 discusses related work. Section 3 outlines the proposed option. Section 4 includes a discussion of the proposed option. Section 5 has results from benchmarking the performance of our implementation. Section 6 is our conclusion.

2.0 Related Work

What makes our TCP RST GUARD option different is its simplicity and backwards compatibility. We list some comparisons of transport layer solutions to RST GUARD below.

The TCP MD5 option's primary purpose is to authenticate a BGP connection [RFC2385]. TCP MD5 uses shared secret keys established out of band prior to connection setup to authenticate the entire TCP segment. This includes the data, TCP header and certain field of the IP header of every packet. Packets are only accepted if the signature included is valid for that packet so an attacker would need to know the secret key in order to mount a TCP RST/FIN attack. TCP MD5 is not widely deployed in Internet routers because it requires shared secret keys, it is computational expensive since it requires a hash be calculated for every packet that is send and received, and it is vulnerability to collision based attacks due to using MD5. Our TCP RST GUARD on the other hand is only computationally expensive for connection setup and teardown, does not required shared secret keys, and uses HMAC-SHA256 to avoid the vulnerabilities to collisions that MD5 and SHA-1 faces.

It is also possible to authenticate using the TCP time stamp option [Poon]. A host will discard a TCP segment if the time stamp of that packet is out of the window, which is defined by the time stamps of other packets received in the same connection. The problem with time stamps is it can be easily predicted if it was just the time on the host, or an attacker can probe the host using unrelated traffic to find it. In TCP RST GUARD the hash values are computed using a secret key for each host so it is much harder to predict since the key can be any length and chosen in any manner as long as it is unpredictable to the attacker.

The other transport layer solutions involve making TCP more complex and severely altering TCP's control process. One solution states that RST packets only get processed if it has the right sequence numbers [Ramaiah]. This same solution also mentions that RST packets must be ACKed and replied with another RST packet with the ACKed sequence number. This solution makes TCP more complex and changes its control process to the point where correctness is an issue. Both are also not backwards compatible while TCP RST GUARD is only used if it is configured in the connection setup keeping it backwards compatible with hosts that do not support the option. Another possible solution is altering the window size to help prevent TCP RST/FIN attacks but this solution requires careful tuning because different applications react differently to specific window sizes and also can be inefficient for connections with a high bandwidth-delay product [Semke].

3.0 Proposal

For our proposed option, the hashing algorithm HMAC-SHA256 is used. The reason for using HMAC is due to the fact it is a strong cryptographic primitive for generating a hash value based on a message and a secret key [RFC2104]. Furthermore SHA-256 is used due to vulnerabilities to collisions in both MD5 and SHA-1.

3.1 Protocol

The following example illustrates the operation of the proposed option. Assume host A and host B want to create a TCP connection that is not vulnerable to RST attacks. For backwards compatibility, if both hosts do not support the option, then the option is turned off. Note that the pipe operator (|) is used to denote concatenation and all values are assumed to be in big endian byte order. Furthermore the notation HMAC-SHA256(x, y) is used to denote applying the HMAC-SHA256 algorithm to the inputs x and y where x is the key and y is the message.

1. Host A and host B each choose a secret key before setting up a connection. A’s secret key is Ka and B’s secret key is Kb. These keys can be any length, chosen in any manner as long as it is unpredictable to an attacker, should not change, and should be persistent across crash failures.

2. Host A chooses a 64 bit nonce value in a unpredictable and uniformly random manner at the start of each connection.

3. Host A calculates it’s secret value as the 64 least significant bits returned by the following function

   SVa = HMAC-SHA256(Ka, nonce | A’s IP | A’s port | B’s IP | B’s Port)

4. Host A then calculates the connection unique hash as the 64 least significant bits of the following function.

   Ca = HMAC-SHA256(SVa, nonce | A’s IP | A’s port | B’s IP | B’s Port)

5. Host A then sends a SYN packet to B with the TCP Reset Guard option specified and with Ca and the nonce value in the header.

6. Host B then calculates its secret value in a manner similar to A, using the nonce value sent by A.

   SVb = HMAC-SHA256(Kb, nonce | B’s IP | B’s port | A’s IP | A’s Port)

7. Host B then calculates it’s connection unique hash in the same manner as A.

   Cb = HMAC-SHA256(SVb, nonce | B’s IP | B’s port | A’s IP | A’s Port)

8. Host B sends a SYN-ACK to A with the TCP Reset Guard option specified and Cb and the nonce value it received from A in the header.

9. In any subsequent packet sent between A and B they must specify the TCP Reset Guard option and include the nonce value in the header. If either side receives a packet with a nonce value that does not match the one the connection was setup with or does not have the TCP Reset Guard option set, then it should be discarded.

10. If either side wants to reset or close the connection they must recalculate their secret value (SVa or SVb) and send it in the header along with the RST or FIN packet. The other side can then use that secret value to calculate the connection hash and verify that it matches the hash they received during the three way handshake.

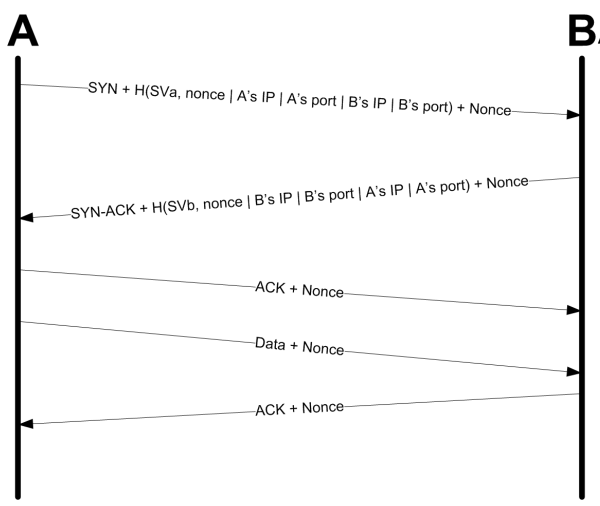
A
B
SYN + H(SVa, nonce | A's IP | A's port | B's IP | B's port) + Nonce
SYN-ACK + H(SVb, nonce | B's IP | B's port | A's IP | A's port) + Nonce
ACK + Nonce
Data + Nonce
ACK + Nonce

Figure 1: Connection Setup

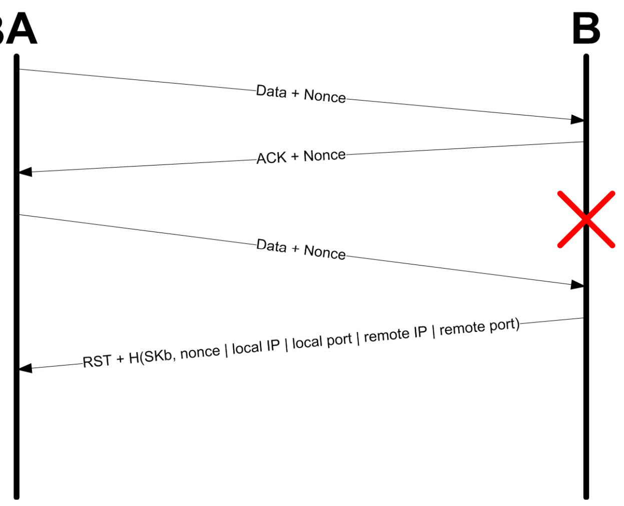
A
B
Data + Nonce
ACK + Nonce
Data + Nonce
RST + H(SKb, nonce | local IP | local port | remote IP | remote port)

Figure 2: Example of a reset after a crash

3.2 Syntax

Our proposal has the following formats for the option's section of the TCP header depending on which phase it is in.

SYN or SYN-ACK packets:

```
+---------+---------+-------------------+
| Kind=29 |Length=18| Connection Hash...|
+---------+---------+-------------------+
|                                       |
+-------------------+-------------------+
|                   | Nonce Value...    |
+-------------------+-------------------+
|                                       |
+-------------------+-------------------+
|                   |
+-------------------+
```

RST or FIN packets:

```
+---------+---------+-------------------+
| Kind=29 |Length=18| Secret Value...   |
+---------+---------+-------------------+
|                                       |
+-------------------+-------------------+
|                   | Nonce Value...    |
+-------------------+-------------------+
|                                       |
+-------------------+-------------------+
|                   |
+-------------------+
```

All other packets:

```
+---------+---------+-------------------+
| Kind=29 |Length=10| Nonce Value...    |
+---------+---------+-------------------+
|                                       |
+-------------------+-------------------+
|                   |
+-------------------+
```

4.0 Discussion

The mechanism proposed provides an effective mechanism to prevent against TCP reset attacks. Using a nonce value ensures that the secret values generated change every time so even if an attacker eavesdropping on the line sees a RST or FIN packet with the secret value, it will be of no use to them in closing future connections. Also using connection identifying information in calculating both the secret value and the connection hash ensures that the secret value produced by a host for a particular nonce is unique to that connection and nonce. The reason 64 bit values are used for the nonce and secret value to make it infeasible to attempt a brute force attack.

Another important attribute of our option is that it is robust if the application or even the host crashes. Since the secret keys are persistent and host specific (i.e. not application specific), when a host receives a packet for which it has no connection for, it can recalculate the secret value from the data in the packet's header fields and then send a RST along with the secret value. This is known as a connectionless reset since it is resetting a connection for which the host has no information about. Also if an organization wants to quickly replace a crashed host with a different one on the same IP, they could have all hosts in the organization use the same secret key.

In comparison to TCP MD5, our mechanism does not incur the any per packet overhead (except to write/verify the nonce value in the header), allows connectionless resets, and does away with the need for shared secret keys. Doing away with shared keys eliminates administrative overhead for organizations to setup agreed upon keys. It also allows any two hosts wishing to have a connection that cannot be arbitrarily reset by a third party to setup such a connection. However, TCP MD5 authenticates the contents of every packet ensuring that the packet was sent by the other host (or someone that knew the secret key). TCP Reset Guard does not do this so an attacker can still get a host to accept arbitrary non RST or FIN packets if they fall in the window. The nonce value used by TCP Reset Guard does however prevent such a sequence number attack from being feasible if the attacker cannot read the contents of packets being sent between the end hosts.

4.1 Security Analysis

In this section we consider potential attacks on an established connection since TCP RST Guard is only intended to prevent spoofed RST or FIN packets for existing connections. We assume an attacker has the IP addresses and port numbers of both ends of the connection as well as the nonce value and the connection hash values. We also assume an attacker may send spoofed packets, but cannot alter packets or filter packets since doing so would give an attacker the means to deny communication between the two hosts using TCP which is the whole purpose of the RST attack.

In order for either host to close an established connection it must be sent a RST/FIN packet with a 64-bit value that when passed in as a key to the HMAC-SHA256 function along with the message constructed from connection identifying information and the nonce value generates the value the host was sent during connection setup (i.e. the connection hash value). It should be noted that an attacker has no control over the connection hash value that was sent during connection setup or the message passed into HMAC-SHA256 since the IP address and port numbers identify to which the connection the packet belongs and packets with incorrect nonce values are discarded. The attacker therefore must be able to determine the secret key using only a known message and the output of HMAC-SHA256. This is not thought to be possible since the key space is too large for a brute for attack (2^64) and because HMAC-SHA256 is not reversible since SHA256 is not reversible. The same argument applies to trying to determine a host's secret key from a known secret value observed when a connection was closed or reset with the caveat that the key space is even larger (2^512).

Since it is not possible for an attacker to calculate an unknown key with a known message and known output of HMAC-SHA256, another possible vector of attack is to send packets to a host that will generate a RST response and see what secret value gets put in the RST packet. While these secret values will not help the attacker determine the host's secret key, it will allow the attacker to build up a database of (source IP, source port, destination IP, destination port, nonce value) to secret value pairs. However since there are 2^64 possible nonce values, creation of such a mapping is not feasible.

4.2 CPU Denial of Service Attack

Since our protocol only addresses RST/FIN attacks, we still fall victim to the denial of service attacks(DOS) that target TCP. In addition to those attacks, our protocol can also lead to a CPU denial of service attack because of the extra strain on the CPU for calculating secret values. The secret values are only calculated during the initial handshake and during sending or receiving a RST/FIN packet. We will not talk about the CPU DOS attacks that occur during the initial handshake because they have been addressed with common defenses before [RFC4987].

An attacker may attempt to send packets with our option enabled to non-existent connections on the host in order to try and get the host to calculate a secret value for the non-existent connection to put in the RST packet. Calculation of both the secret values and connection hash values takes around 10,000 cycles which for comparison was about the time of two context switches our system.

5.0 Evaluation

To measure the benefit of the TCP RST GUARD option, we wrote a benchmark suite that computes the round trip time and setup and teardown time for a TCP connection. We compare a TCP connection with the TCP RST GUARD option set to a connection with no options set and one with the TCP MD5 option set.

5.1 Experiment Description

We modified the TCP IPv4 stack of the 2.6.29 release of the Linux kernel to add in the RST GUARD option. We built and ran the modified version of the Linux kernel on a 2.5 GHz Core 2 Duo for the server and a 2.33 GHz Core 2 Duo machine for the client. To run our benchmark suite, we connected the machines with a CAT 6 crossover cable. We used Intel's RDTSC instruction to read the processors cycle counter to get our timing data.

For the round trip time benchmark, we measure the time to send 1 byte over the network and receive a reply. The benchmark runs for 10,000 iterations and calculates the average round trip time. The setup and teardown benchmark establishes a connection between the client and server and then closes the connection, without sending any bytes over the network. The benchmark runs for 50 iterations and calculates the average setup and teardown time.

5.2 Results

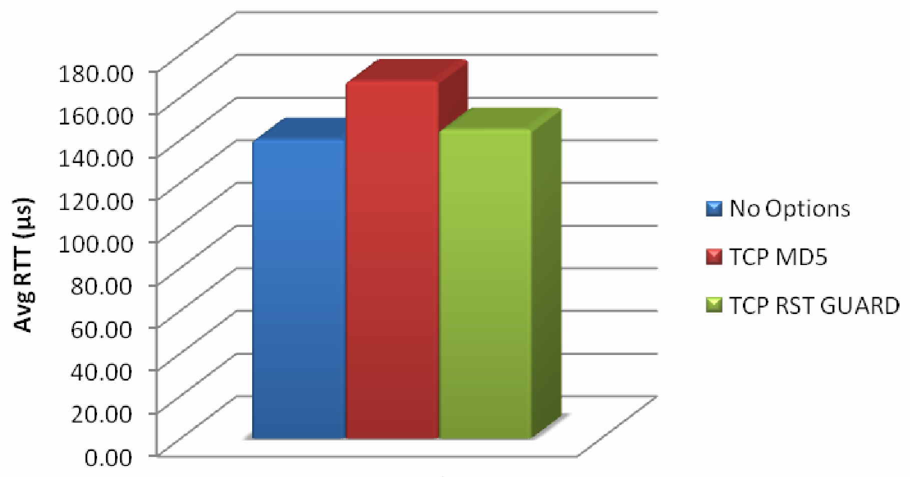

Figure 3: Average Round Trip Time

We first explore the difference in round trip time between the three types of TCP connections. Figure 3 shows the round trip time for a connection with no options set, a connection with the TCP MD5 option set, and a connection with the RST GUARD option set in microseconds. Significantly, Figure 3 shows that the TCP connection with the TCP MD5 option set takes an additional 20 more microseconds than a connection with the TCP RST GUARD option set and 25 more microseconds than a connection with no options set. The TCP connection with no options set was faster than both a connection with either TCP MD5 or RST GUARD set because the connection with no options takes the fast path while both the RST GUARD option and the TCP MD5 option take the slow path. However, the RST GUARD option could be modified to take the fast path since there's no processing for an established connection other than verifying the nonce value. The connection with the TCP MD5 option set was significantly slower than the connection with the RST GUARD option set because the RST GUARD option only needs to write the nonce to the header when sending packets and checking the nonce value when receiving packets, but the TCP MD5 option needs to calculate the MD5 hash of the packet for every packet sent and received. The RST GUARD option is much more efficient and equally effective as the TCP MD5 option in preventing RST attacks, but requires no per-packet calculations and the differences between TCP MD5 and TCP RST GUARD would be even more pronounced on slower processors.

Table 1: Experimental Results (times in µ seconds)

Test	No Options	TCP MD5	TCP RST GUARD
Avg RTT	140.04	167.41	144.86
Setup Time	177.09	185.01	207.76
Teardown Time	3.30	4.03	6.94

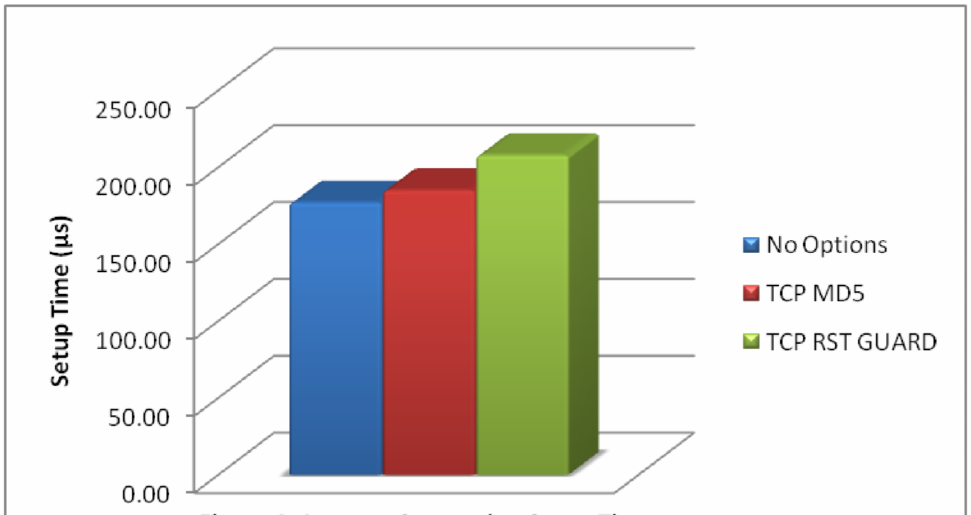

Figure 4: Average Connection Setup Time

Next we explore the difference in setup and teardown times between the three types of connections. Figure 4 and Figure 5 show the setup and teardown time for a connection with no options set, a connection with the TCP MD5 option set, and a connection with the RST GUARD option set in microseconds. Figure 4 shows that the TCP connection with the RST GUARD option set takes 20 additional microseconds to setup a connection than the TCP MD5 option and 30 more microseconds than a connection with no options. The RST GUARD option takes a slightly longer to setup than both the connection with no options and the TCP MD5 option because the RST GUARD option has to get a random nonce value and each side must also calculate their secret value and the connection unique hash value.

Figure 5 shows that the RST GUARD option takes only 3 additional microseconds to teardown a connection than the TCP MD5 option and 4 more microseconds than a connection with no options. The teardown time only measures the time from invoking the teardown method to when the packet gets put on the queue to be sent out because the kernel will return back to the program before the packet actually gets sent. Furthermore in our implementation the host recalculates the secret value, but an easy optimization would be to simply store the secret value calculated during the three-way handshake and just write that to the header which should make the performance comparable to no options. The additional time needed to setup and teardown a connection with the RST GUARD option set does not affect the overall efficiency as much compared to needing additional time to send and receive packets over the network because setup and teardown are only required once per connection. Also on a typical internet link where the round trip time is measured in milliseconds, an extra 30 microseconds of overhead during the three-way handshake will not even be noticeable.

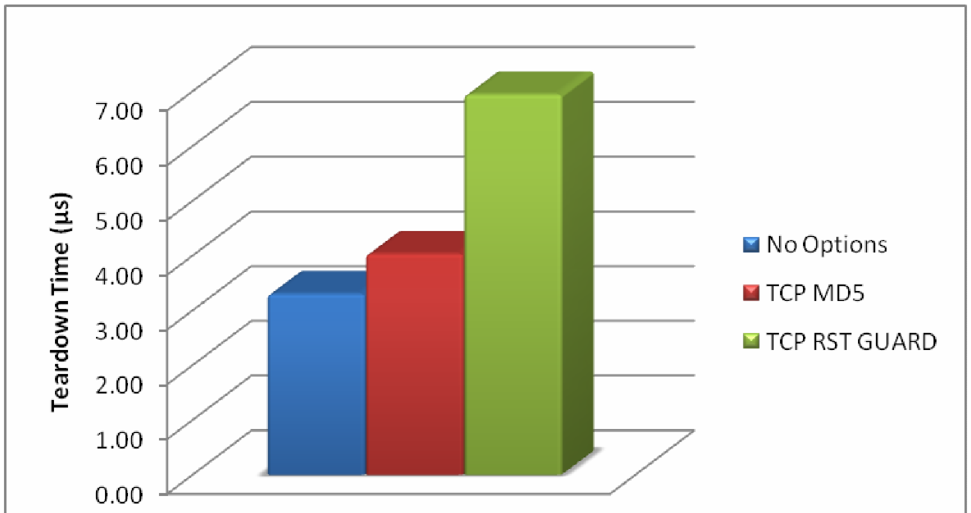

Figure 5: Average Connection Teardown Time

6.0 Conclusion

We have proposed a new mechanism for preventing TCP RST attacks. Our mechanism improves on other mechanism like ACKing RSTs or using smaller windows because it is backwards compatible and doesn't introduce any constraints on the functioning of TCP or correctness of programs using these mechanisms. Furthermore, in comparison to TCP MD5, our mechanism incurs no per-packet overhead, does not require shared secret keys, and allows connectionless resets.

7.0 References

[RFC2385] Heffernan, A., "Protection of BGP Sessions via the TCP MD5 Signature," RFC 2385, August 1998.

[Watson] Watson, Paul, "Slipping in the Window: TCP Reset Attacks", October 2003.

[RFC2104] Krawczyk, et. al., "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, February 1997

[Poon] Poon, K., "Use of TCP timestamp option to defend against blind spoofing attack", Work in Progress, October 2004.

[Ramaiah] Ramaiah, A., Stewart, R., and M. Dalal, "Improving TCP's Robustness to Blind In-Window Attacks", Work in Progress, July 2007.

[Semke] Semke, J., J. Mahdavi, and M. Mathis, "Automatic TCP Buffer Tuning", ACM SIGCOMM '98/ Computer Communication Review, volume 28, number 4, Oct. 1998.

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