TCP Congestion Control with a Misbehaving Receiver

Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson
Department of Computer Science and Engineering
University of Washington, Seattle

Abstract

In this paper, we explore the operation of TCP congestion control when the receiver can misbehave, as might occur with a greedy Web client. We first demonstrate that there are simple attacks that allow a misbehaving receiver to drive a standard TCP sender arbitrarily fast, without losing end-to-end reliability. These attacks are widely applicable because they stem from the sender behavior specified in RFC 2581 rather than implementation bugs. We then show that it is possible to modify TCP to eliminate this undesirable behavior entirely, without requiring assumptions of any kind about receiver behavior. This is a strong result: with our solution a receiver can only *reduce* the data transfer rate by misbehaving, thereby eliminating the incentive to do so.

1 Introduction

End-to-end congestion control mechanisms, such as those used in TCP, are the primary means used for sharing scarce bandwidth resources in the Internet. These mechanisms implicitly rely on both endpoints to cooperate in determining the proper rate at which to send data. Obviously, if the sending endpoint misbehaves, and does not obey the appropriate congestion control algorithms, then it may send data more quickly than well-behaved hosts – possibly forcing competing traffic to be delayed or discarded. Less obviously, a misbehaving *receiver* can achieve the same result.

While the possibility of such attacks has been hinted at previously [APS99, PAD$^+$99], we believe the ease of exploiting this vulnerability and the potential impact are not fully appreciated. We note that the population of receivers is extremely large (all Internet users) and has both the incentive (faster Web surfing) and the opportunity (open source operating systems) to exploit this vulnerability.

In this paper, we explore the impact that a misbehaving receiver can have on TCP congestion control. We present two kinds of results. First, we identify several vulnerabilities that can be exploited by a malicious receiver to defeat TCP congestion control. This can be done in a manner that does not break end-to-end reliability semantics and that relies only on the standard behavior of correctly implemented TCP senders. Tests against live Web servers using a modified TCP implementation that we produced for this purpose, "TCP Daytona", confirm that common TCP implementations possess these vulnerabilities. Second, we show that it is possible to modify the design of TCP to eliminate this behavior – without requiring that the receiver be trusted in any manner. With our protocol modifications, a faulty or malicious receiver can at most cause the sender to transmit data at a slower rate than it otherwise would, thus harming only itself. Because our work has serious practical ramifications for an Internet that depends on trust to avoid congestion collapse, we also describe backwards-compatible mechanisms that can be implemented at the sender to mitigate the effects of untrusted receivers.

As far as we are aware, the division of trust between sender and receiver has not been studied previously in the context of congestion control. While end-to-end congestion control protocols assume that both sender and receiver behave correctly, in many environments the interests of sender and receiver may differ considerably – creating significant incentives to violate this "good faith" doctrine. For example, in many wide-area data retrieval applications, such as Web browsing, the sender's interest is to provide uniform service to all clients requesting data, while the interest of each client receiver is to maximize its own data throughput. Traditional cryptographic security mechanisms, such as embodied in IPSEC [KA98], can provide guarantees of authentication and confidentiality, but they can not prevent a receiver from violating TCP's congestion control specification and consequently undermining the fairness and stability provided therein.

The potential congestion resulting from aggressive *senders* has received significant attention from the networking community [She94, FF99, RHE99, VRC98], and has produced proposals for per-flow bandwidth reservation [ZDE$^+$93] and mechanisms to detect and limit "unfriendly" flows in the network [FF99]. These solutions, if workable, would solve the more general problem of unconstrained data transmission and would make the issue of trust in end-to-end congestion control less pressing. However, given that it is unlikely that such mechanisms will be widely deployed in the near term, we feel it is still prudent to consider the potential impact of untrusted receivers on the congestion control mechanisms in today's Internet.

2 Vulnerabilities

By systematically considering sequences of message exchanges, we have been able to identify several vulnerabilities that allow misbehaving receivers to control the sending rate of unmodified, conforming TCP senders. This section describes these vulnerabilities and techniques for exploiting them. In addition to denial-of-service attacks, these techniques can be used to enhance the performance of the attacker's TCP sessions at the expense of behaving clients.

This work was funded by generous grants from NSF (CCR 94-53532), DARPA (F30602-98-1-0205), USENIX, the National Library of Medicine, Cisco, Fuji and Intel. Correspondence concerning this paper may be sent to savage@cs.washington.edu.

2.1 TCP review

While a detailed description of TCP's error and congestion control mechanisms is beyond the scope of this paper, we describe the rudiments of their behavior below to allow those unfamiliar with TCP to understand the vulnerabilities explained later. For simplicity, we consider TCP without the Selective Acknowledgment option (SACK) [MMFR96], although the vulnerabilities we describe also exist when SACK is used.

TCP is a connection-oriented, reliable, ordered, byte-stream protocol with explicit flow control. A sending host divides the data stream into individual segments, each of which is no longer than the Sender Maximum Segment Size (SMSS) determined during connection establishment. Each segment is labeled with explicit sequence numbers to guarantee ordering and reliability. When a host receives an in-sequence segment it sends a cumulative acknowledgment (ACK) in return, notifying the sender that all of the data preceding that segment's sequence number has been received and can be retired from the sender's retransmission buffers. If an out-of-sequence segment is received, then the receiver acknowledges the next contiguous sequence number that was *expected*. If outstanding data is not acknowledged for a period of time, the sender will timeout and retransmit the unacknowledged segments.

TCP uses several algorithms for congestion control, most notably *slow start* and *congestion avoidance* [Jac88, Ste94, APS99]. Each of these algorithms controls the sending rate by manipulating a congestion window (*cwnd*) that limits the number of outstanding unacknowledged bytes that are allowed at any time. When a connection starts, the slow start algorithm is used to quickly increase *cwnd* to reach the bottleneck capacity. When the sender infers that a segment has been lost it interprets this has an implicit signal of network overload and decreases *cwnd* quickly. After roughly approximating the bottleneck capacity, TCP switches to the congestion avoidance algorithm which increases the value of *cwnd* more slowly to probe for additional bandwidth that may become available.

We now describe three attacks on this congestion control procedure that exploit a sender's vulnerability to non-conforming receiver behavior.

2.2 ACK division

TCP uses a byte granularity error control protocol and consequently each TCP segment is described by sequence number and acknowledgment fields that refer to byte offsets within a TCP data stream. However, TCP's congestion control algorithm is implicitly defined in terms of segments rather than bytes. For example, the most recent specification of TCP's congestion control behavior, RFC 2581, states:

> 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).

The incongruence between the byte granularity of error control and the segment granularity (or more precisely, SMSS granularity) of congestion control leads to the following vulnerability:

> **Attack 1**:
> *Upon receiving a data segment containing N bytes, the receiver divides the resulting acknowledgment into M, where $M \leq N$, separate acknowledgments – each covering one of M distinct pieces of the received data segment.*

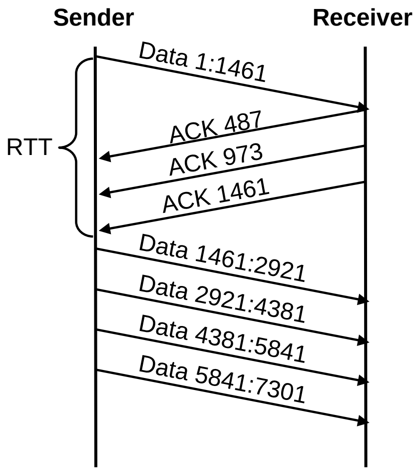

Figure 1: Sample time line for a ACK division attack. The sender begins with *cwnd*=1, which is incremented for each of the three valid ACKs received. After one round-trip time, *cwnd*=4, instead of the expected value of *cwnd*=2.

This attack is demonstrated in Figure 1 with a time line. Here, each message exchanged between sender and receiver is shown as a labeled arrow, with time proceeding down the page. The labels indicate the type of message, data or acknowledgment, and the sequence space consumed. In this example we can see that each acknowledgment is valid, in that it covers data that was sent and previously unacknowledged. This leads the TCP sender to grow the congestion window at a rate that is M times faster than usual. The receiver can control this rate of growth by dividing the segment at arbitrary points – up to one acknowledgment per byte received (when M = N). At this limit, a sender with a 1460 byte SMSS could *theoretically* be coerced into reaching a congestion window in excess of the normal TCP sequence space (4GB) in only four round-trip times! [1] Moreover, while high rates of additional acknowledgment traffic may increase congestion on the path to the sender, the penalty to the receiver is negligible since the cumulative nature of acknowledgments inherently tolerates any losses that may occur.

2.3 DupACK spoofing

TCP uses two algorithms, *fast retransmit* and *fast recovery*, to mitigate the effects of packet loss. The fast retransmit algorithm detects loss by observing three duplicate acknowledgments and it immediately retransmits what appears to be the missing segment. However, the receipt of a duplicate ACK also suggests that segments are leaving the network. The fast recovery algorithm employs this information as follows (again quoted from RFC 2581):

> 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.

[1] Of course the practical transmission rate is ultimately limited by other factors such as sender buffering, receiver buffering and network bandwidth.

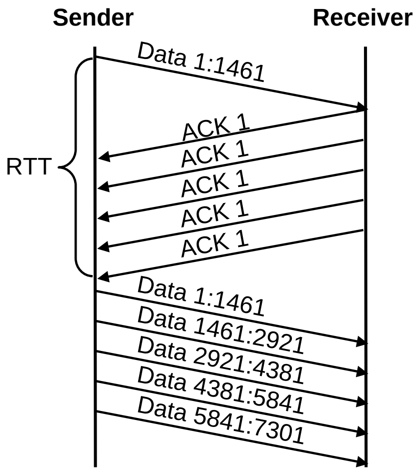

Figure 2: Sample time line for a DupACK spoofing attack. The receiver forges multiple duplicate ACKs for sequence number 1. This causes the sender to retransmit the first segment and send a new segment for each additional forged duplicate ACK.

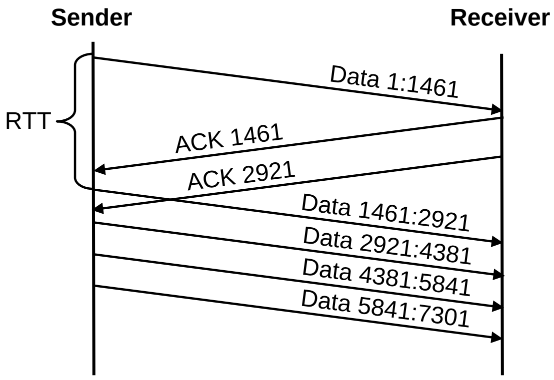

Figure 3: Sample time line for optimistic ACKing attack. The ACK for the second segment is sent before the segment itself is received, leading the receiver to grow *cwnd* more quickly than otherwise. At the end of this example, *cwnd*=3, rather than the expected value of *cwnd*=2.

There are two problems with this approach. First, it assumes that each segment that has left the network is full sized – again an unfortunate interaction of byte granularity error control and segment granularity congestion control. Second, and more important, because TCP requires that duplicate ACKs be *exact* duplicates, there is no way to ascertain which data segment they were sent in response to. Consequently, it is impossible to differentiate a “valid” duplicate ACK, from a forged, or “spoofed”, duplicate ACK. For the same reason, the sender cannot distinguish ACKs that are accidentally duplicated by the network itself from those generated by a receiver [APS99]. In essence, duplicate ACKs are a signal that can be used by the receiver to force the sender to transmit new segments into the network as follows:

> **Attack 2**:
> *Upon receiving a data segment, the receiver sends a long stream of acknowledgments for the last sequence number received (at the start of a connection this would be for the SYN segment).*

Figure 2 shows a time line for this technique. The first four ACKs for the same sequence number cause the sender to retransmit the first segment. However, *cwnd* is now set to its initial value plus 3*SMSS, and increased by SMSS for each additional duplicate ACK, for a total of 4 segments (as per the fast recovery algorithm). Since duplicate ACKs are indistinguishable, the receiver does not need to wait for new data to send additional acknowledgments. As a result, the sender will return data at a rate directly proportional to the rate at which the receiver sends acknowledgments. After a period, the sender will timeout. However, this can easily be avoided if the receiver acknowledges the missing segment and enters fast retransmit again for a new, later, segment.

2.4 Optimistic ACKing

Implicit in TCP's algorithms is the assumption that the time between a data segment being sent and an acknowledgment for that segment returning is at least one round-trip time. Since TCP's congestion window growth is a function of round-trip time (an exponential function during slow start and a linear function doing congestion avoidance), sender-receiver pairs with shorter round-trip times will transfer data more quickly.

However, the protocol does not use any mechanism to enforce its assumption. Consequently, it is possible for a receiver to *emulate* a shorter round-trip time by sending ACKs optimistically for data it has not yet received:

> **Attack 3**:
> *Upon receiving a data segment, the receiver sends a stream of acknowledgments anticipating data that will be sent by the sender.*

This technique is demonstrated in Figure 3. Note that while it is easy for the receiver to anticipate the correct sequence numbers to use in each acknowledgment (since senders generally send full-sized segments), this accuracy is not necessary. As long as the receiver acknowledges new data the sender will transmit additional segments. Moreover, if an ACK arrives for data that has not yet been sent, this is generally ignored by the sending TCP – allowing a sender to be arbitrarily aggressive in its generation of optimistic ACKs.

Unlike the previous attacks, this technique does not necessarily preserve end-to-end reliability semantics – if data from the sender is lost it may be unrecoverable since it has already been acknowledged. However, new features in protocols such as HTTP-1.1 allow receivers to request particular byte-ranges within a data object [FGM+99]. This suggests a strategy in which data is gathered on one connection and lost segments are then collected selectively with application-layer retransmissions on another. Optimistic ACKing could be used to ramp the transfer rate up to the bottleneck rate immediately, and then hold it there by sending acknowledgments in spite of losses. This ability of the receiver to conceal losses is extremely dangerous because it eliminates the only congestion signal available to the sender. A malicious attacker could conceal *all* losses and therefore lead a sender to increase *cwnd* indefinitely – possibly overwhelming the network with useless packets.

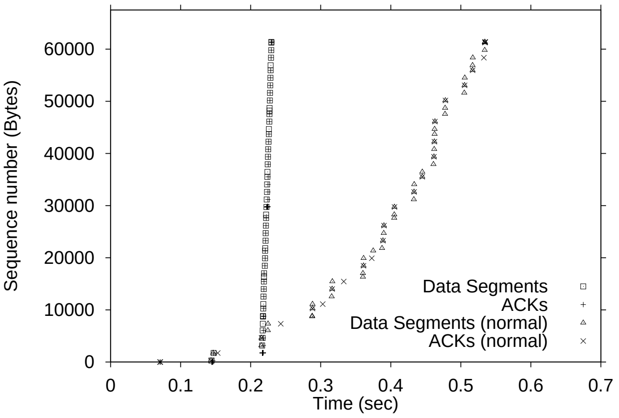

Figure 4: The TCP Daytona *ACK division* attack convinces the TCP sender to send all but the first few segments of a document in a single burst.

3 Implementation experience

To exploit the vulnerabilities described above, we made three modifications to the TCP subsystem of Linux 2.2.10. This resulting TCP implementation, which we refer to facetiously as "TCP Daytona", provides extremely high performance at the expense of its competitors. We demonstrate these abilities with time sequence plots of packet traces for both normal and modified receiver TCP's. Needless to say, our implementation is intentionally not "stable", and would likely lead to congestion collapse if it were widely deployed.

3.1 ACK division

The TCP Daytona ACK division algorithm adds 24 lines of code that divide each new outgoing ACK into many ACKs for smaller extents of the sequence space. Half of the new code is dedicated to ensuring that the number of outgoing ACKs is no more than should be needed to coerce a sender in slow start to saturate our test machine's 100Mbps Ethernet interface.

Figure 4 shows client-side TCP sequence number plots of our test machine making an HTTP request for the `index.html` object from `cnn.com`, with and without our ACK division attack enabled. This figure spans the entire transaction, beginning with the TCP handshake that starts at 0ms and ends at around 70ms, when the HTTP request is sent. The first HTTP data from the server arrives at around 140ms.

This figure shows that, when this attack is enabled, the many small ACKs sent around 140ms convince the Web server to unleash the entire remainder of the document in a single burst; this data arrives exactly one round-trip time later. By contrast, with the normal TCP implementation, the server spreads out the data over the next four round-trip times. In general, as this figure suggests, this attack can convince a TCP sender to send all of its data in a single burst.

3.2 DupACK spoofing

The TCP Daytona DupACK spoofing attack is implemented by 11 lines of code that cause the receiver to send sufficient duplicate ACKs such that the sender (re-)enters fast recovery and fills the receiver's advertised flow control window each round-trip time.

Figure 5 shows another client-side plot of the same HTTP request, this time with the DupACK spoofing attack superimposed

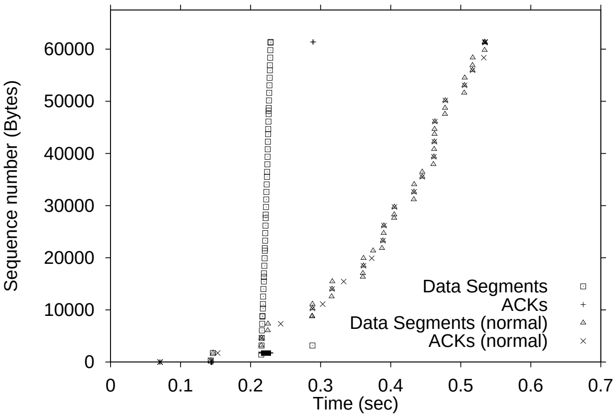

Figure 5: The TCP Daytona *DupACK spoofing* attack, like the ACK division attack, convinces the TCP sender to send all but the first few segments of a document in a single burst.

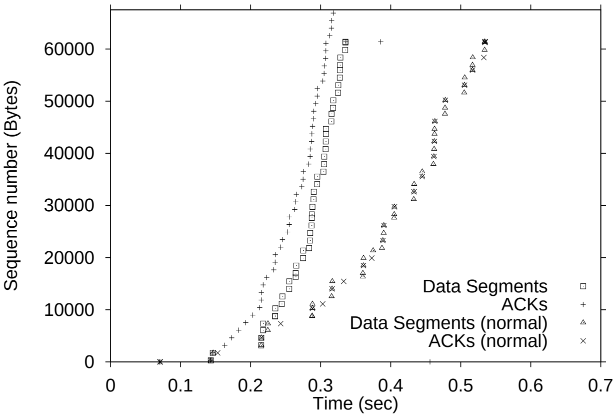

Figure 6: The TCP Daytona *optimistic ACK* attack, by sending a stream of early ACKs, convinces the TCP sender to send data much earlier than it normally would.

on a normal transfer. The many duplicate ACKs that the receiver sends at around 140ms cause the sender to enter fast recovery and transmit the rest of the data, which arrives at around 210ms. Were there more data, the flurry of duplicate ACKs sent at 210ms-230ms would elicit another burst from the sender. Since there is no more new data, the sender simply fills in the hole it perceives; this segment arrives at around 290ms. This figure illustrates how the DupACK spoofing attack can achieve performance essentially equivalent to the ACK division attack – namely, both attacks can convince the sender to empty its entire send buffer in a single burst.

3.3 Optimistic ACKing

The TCP Daytona implementation of optimistic ACKing consists of 45 lines of code. Because acknowledging data that has not arrived is a fundamentally tricky business, we chose a very simple implementation as a proof of concept. When a TCP connection for an HTTP or FTP client receives its first data, we set a timer to expire every 10ms. Any interval would do, but we chose 10ms because it is the smallest interval that Linux 2.2.10 supports on the Intel PC platform. Whenever this periodic timer expires, or a new data segment arrives, our receiver sends a new optimistic ACK for one MSS beyond the previous optimistic ACK.

Figure 6 shows our optimistic ACK algorithm in action transferring the same `index.html`, again with a normal transfer superimposed. Note that after the first few data segments arrive at around 140ms, the receiver sends a steady stream of ACKs, where each ACK is sent about 10ms-70ms *before* the corresponding data arrives! The result is that the data transfer employing optimistic ACKs completes in approximately half the normal transfer time. Though this is a modest gain relative to the other attacks, a more bold optimistic ACKing scheme could achieve far greater throughput by acknowledging data at a more rapid pace.

3.4 Applicability

In order to verify that common TCP implementations have these vulnerabilities, we tested each attack against a set of nine Web servers running a diverse array of popular server operating systems. To determine the operating system running on each Web server, we used `nmap`, a tool that identifies TCP implementations based on the "fingerprint" of their characteristic response to TCP segments whose correct response is under-specified [Vas]. To further decrease the possibility of misidentifying an operating system, in all but three cases we were able to use Web servers that were operated by OS vendors and that `nmap` confirmed were running the high-end server operating system from that vendor.

Table 1 shows which TCP implementations are vulnerable to each attack. The attacks are all widely applicable, with three exceptions. First, Linux 2.2 is not vulnerable to the ACK division attack because it increases its congestion window only if at least one whole previously-unacknowledged segment is acknowledged. Second, Linux 2.0 refuses to count duplicate acknowledgments until *cwnd* is greater than three. Consequently, the DupACK attack will fail if initiated on connection startup. Finally, Windows NT appears to have a bug that causes it to rarely, if ever, enter fast recovery. This bug renders NT immune to attacks that rely on extra duplicate acknowledgments.

4 Solutions

As demonstrated in the previous section, TCP's current specification has several vulnerabilities that allow a misbehaving receiver to control the sender's transmission rate. While it is impossible to *force* a receiver to behave correctly, it is both possible and desirable to remove its incentive to misbehave. That is, we wish to ensure that a misbehaving receiver can not obtain data faster than a behaving one. In this section we describe simple modifications to the TCP protocol that, without changing the nature of congestion control, allow the verification of what has historically been an implicit contract between sender and the receiver – that each acknowledgment faithfully and unambiguously reflects data that has been successfully transferred to the receiver.

4.1 Designing robust protocols

We believe TCP's vulnerabilities arise from a combination of unstated assumptions, casual specification and a pragmatic need to develop congestion control mechanisms that are backward compatible with previous TCP implementations. In retrospect, if the contract between sender and receiver had been defined explicitly these vulnerabilities would have been obvious.

We are inspired by Abadi and Needham's paper, *Prudent Engineering Practice for Cryptographic Protocols*, which presents a set of design rules that are surprisingly germane to this problem [AN96]. In particular we reprint their first three principles below:

Principle 1. Every message should say what it means: the interpretation of the message should depend only on its content.

Principle 2. The conditions for a message to be acted upon should be clearly set out so that someone reviewing a design may see whether they are acceptable or not.

Principle 3. If the identity of a principal is essential to the meaning of a message, it is prudent to mention the principal's name explicitly in the message.

4.2 ACK division

This vulnerability arises from an ambiguity about how ACKs should be interpreted – a violation of the second principle. TCP's error-control allows an ACK to specify an arbitrary byte offset in the sequence space while the congestion control specification assumes that an ACK covers an entire segment.

There are two obvious solutions: either modify the congestion control mechanisms to operate at byte granularity or guarantee that segment-level granularity is always respected. The first solution is virtually identical to the "byte counting" modifications to TCP discussed in [All98, All99]. If *cwnd* is not incremented by a full SMSS, but only proportional to the amount of data acknowledged, then ACK division attacks will have no effect. The second, perhaps simpler, solution is to only increment *cwnd* by one SMSS when a valid ACK arrives that covers the entire data segment sent. As mentioned earlier, this technique is employed in the latest versions of Linux (2.2.x) at the time of this writing.

4.3 DupACK spoofing

During fast recovery and fast retransmit, TCP's design violates the first principle – the meaning of a duplicate ACK is implicit, dependent on previous context, and consequently difficult to verify.

TCP assumes that all duplicate ACKs are sent in response to unique and distinct segments. This assumption is unenforceable without some mechanism for identifying the data segment that led to the generation of each duplicate ACK. The traditional method for guaranteeing association is to employ a *nonce* [Sch96]. We present a simple version of such a nonce protocol below (we will extend it shortly):

> **Singular Nonce**:
> *We introduce two new fields into the TCP packet format: Nonce and Nonce reply. For each segment, the sender fills the Nonce field with a unique random number generated when the segment is sent. When a receiver generates an ACK in response to a data segment, it echoes the nonce value by writing it into the Nonce Reply field.*

The sender can then arrange to only inflate *cwnd* in response to duplicate ACKs whose Nonce Reply value corresponds to a data segment previously sent and not yet acknowledged.

We note that the singular nonce, as we have described it so far, is similar to the Timestamps option [JBB92], with two important differences. First, the Nonce field preserves association for duplicate ACKs, while the Timestamps option does not (preferring instead to reuse the previous timestamp value). Second, and more important, because Timestamps is a *option*, a receiver has the choice to not participate in its use. We cannot rely on misbehaving clients to voluntarily participate in their own policing. For the same reason, we cannot rely on other TCP options, such as proposed extensions to SACK [FMM$^+$99], to eliminate this vulnerability.

	ACK Division	DupACK Spoofing	Optimistic Acks
Solaris 2.6	Y	Y	Y
Linux 2.0	Y	Y (**N**)	Y
Linux 2.2	**N**	Y	Y
Windows NT4/95	Y	**N**	Y
FreeBSD 3.0	Y	Y	Y
DIGITAL Unix 4.0	Y	Y	Y
IRIX 6.x	Y	Y	Y
HP-UX 10.20	Y	Y	Y
AIX 4.2	Y	Y	Y

Table 1: For each TCP Daytona attack, we denote with a "Y" those operating systems that we found to be vulnerable. Most operating systems we tested were vulnerable to all the Daytona attacks.

Unfortunately, our fix requires the modification of clients and servers and the addition of a TCP field. While it is the only complete solution we have discovered, there are sender-only heuristics which can mitigate, although not eliminate, the impact of the DupACK spoofing attack in a purely backward compatible manner. In particular, the sender can maintain a count of outstanding segments sent above the missing segment. For each duplicate acknowledgment this count is decremented and when it reaches zero any additional duplicate acknowledgments are ignored. This simple fix appears to limit the number of segments wrongly sent to contain no more than $cwnd - SMSS$ bytes. Unfortunately, a clever receiver can acknowledge the missing segment and then repeat the process indefinitely unless other heuristics are employed to penalize this behavior (e.g. by refusing to enter fast retransmit multiple times in a single window as suggested in [Flo95]).

4.4 Optimistic ACKing

The optimistic ACK attack is possible because ACKs do not contain any proof regarding the identity of the data segment(s) that caused them to be sent. In the context of the third principle described earlier, a data segment is a principal and an ACK is the message of concern.

This problem is also well addressed using a nonce. If a nonce can't be guessed by the receiver, than ACKs with valid nonces imply that a full round-trip time has taken place (man-in-the-middle attacks notwithstanding).

However, the singular nonce we have described is imperfect because it does not mirror the cumulative nature of TCP. Acknowledgments can be delayed or lost, yet the cumulative property of TCP's sequence numbers ensures that the most recent ACK can cover all previous data. In contrast, the singular nonce only provides evidence that a single segment was received. A misbehaving sender could still mount a denial of service attack by concealing lost data, yet still sending back ACKs with valid nonces. To address this deficiency we describe a *cumulative nonce* as follows:

> **Cumulative Nonce**:
> *For each segment, the sender fills the Nonce field with a unique random number generated when the segment is sent. Each side maintains a nonce sum representing the cumulative sum of all in-sequence acknowledged nonces. When a receiver receives an in-sequence segment it adds the value contained in its Nonce field to this sum. When a receiver generates an ACK in response to a data segment, it either echoes the current value of the nonce sum (for in-sequence data) or echoes the nonce value sent by the sender (for out-of-sequence data).*

The sender can then efficiently verify that the data acknowledged by the receiver has, in fact, been successfully transferred.

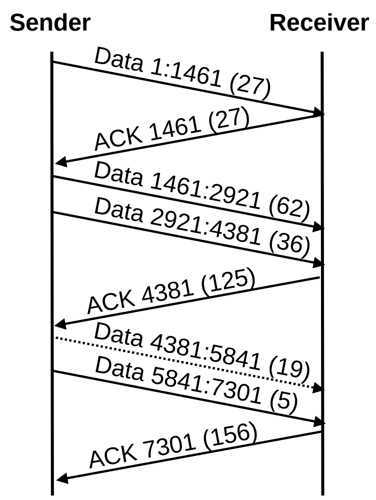

Figure 7: A time line for a transfer using a cumulative nonce. The nonce values are shown in parenthesis and it is assumed that each side starts with a nonce sum of zero. The dotted line indicates a data segment that was dropped. The final ACK, which attempts to conceal the loss of this segment, will be rejected because its cumulative nonce value is incorrect (156 instead of the expected value of 149).

An example of this protocol is depicted in Figure 7. The second ACK (acknowledging bytes 4380 and below) demonstrates the cumulative effect of the nonce, proving that the receiver has in fact seen all three segments (125=27+62+36). The fourth data segment is lost (indicated by a dotted line) and the third ACK attempts to conceal this loss by acknowledging a later segment. However, the ACK will be rejected by the sender, since it cannot provide the proper nonce sum (149) for the data it purports to acknowledge.

A potential complication can occur if the segment boundaries differ between the initial transmission and a subsequent retransmission. Such an occurrence might occur during dynamic path MTU changes. There are several implementation strategies to address this situation, but the simplest is to randomly subdivide the original nonce value in a way that the sum of the new nonce values is still consistent with the original transmission. For example, if a 1460 byte segment is initially transmitted with a nonce value of 14, but subsequent retransmissions are limited to 536 bytes by a path MTU change, then we might retransmit the data in three packets, with nonce values of 7, 3 and 4.

While it is difficult to prevent loss concealment without a cumulative nonce, there are interim sender-side modifications that can approximate a singular nonce and thereby limit the impact of optimistic ACKing attacks. If the sending TCP randomly varies the size of outgoing segments by a small amount (e.g. [SMSS-15 bytes .. SMSS bytes]), a misbehaving receiver will be unable to correctly anticipate the segment boundaries. Consequently, the exact segment boundaries encode a form of nonce and the sending TCP can filter out optimistic ACKs as those that do not fall on the appropriate sequence numbers (this assumes that receivers acknowledge all of the data they receive). As an added disincentive, the sender could send a RST for any ACK that acknowledged data not yet sent. This strategy does not prevent the receiver from concealing loss, but it can mitigate the effects of optimistic ACKs (which we believe is a more attractive attack for the average user).

5 Conclusion

In this paper we have described how a receiver can manipulate the TCP congestion control function managed by the sender, and how the sender can prevent these manipulations. Our work highlights two results that we believe are significant yet not widely appreciated:

- TCP, which was originally designed for a cooperative environment, contains several vulnerabilities that an unscrupulous receiver can exploit to obtain improved service at the expense of other network clients or to implement a denial-of-service attack. We have described ACK division, DupACK spoofing and Optimistic ACK mechanisms and implemented them to demonstrate that the attacks are both real and widely applicable.

- The design of TCP can be modified, without changing the nature of the congestion control function, to eliminate these vulnerabilities. We have described the workings of a new Cumulative Nonce approach that accomplishes this in a simple yet effective manner. We have also identified and described sender-only modifications that can be deployed immediately to reduce the scope of the vulnerabilities without receiver-side modifications.

Our work can readily be extended to other protocols. While the Cumulative Nonce was defined in the context of TCP, it could be adapted to any sender-based congestion control scheme. This might prove fruitful for unreliable transports, for example, either those that are explicitly TCP-friendly, such as RAP [RHE99], or other rate adaptive mechanisms, like those employed by RealAudio. A Cumulative Nonce could also be used more widely to aid in the design of other kinds of protocols. This is because it effectively defines a sequencing mechanism between untrusted parties that, because it is lightweight, idempotent and cumulative, is well suited to network environments.

Beyond these immediate results, our work raises more speculative protocol design issues. TCP was originally designed for a cooperative environment, and its evolution through the years has built on this base. Given this, it is perhaps not so surprising that we were able to find the vulnerabilities we did, because they naturally arise when the sender and receiver represent different interests. With the growth of the Internet, however, it is arguable that "separate interests" should be assumed by default. Protocol functions that are managed by one party would then be designed to minimize the trust they place in other parties. We observe that this kind of "separation of interests" will require new mechanisms, such as a Cumulative Nonce, to guarantee that different parties respect a common behavioral contract.

Acknowledgments

This paper has benefited from conversations with many different people. In particular, we'd like to thank Sally Floyd, Vern Paxson, Jeff Mogul, Greg Minshall, Venkat Padmanabhan and Robert Grimm for their careful reading and thoughtful comments. In addition, Roch Guerin and the anonymous CCR reviewers provided valuable feedback on the submitted version of this paper.

References

[All98] Mark Allman. On the generation and use of TCP acknowledgments. *Computer Communications Review*, 28(5), October 1998.

[All99] Mark Allman. TCP byte counting refinements. *Computer Communications Review*, 29(3), July 1999.

[AN96] Martin Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1), January 1996.

[APS99] M. Allman, V. Paxson, and W. Stevens. TCP congestion control. RFC 2581, April 1999.

[FF99] Sally Floyd and Kevin Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking*, August 1999.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.

[Flo95] Sally Floyd. TCP and successive fast retransmits. `http://www.aciri.org/floyd/papers/fastretrans.ps`, May 1995.

[FMM+99] Sally Floyd, Jamshid Mahdavi, Matt Mathis, Matthew Podolsky, and Allyn Romanow. An extension to the selective acknowledgment (SACK) option for TCP. Internet Draft, August 1999.

[Jac88] Van Jacobson. Congestion avoidance and control. *SIGCOMM '88*, August 1988.

[JBB92] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. RFC 1323, May 1992.

[KA98] S. Kent and R. Atkinson. Security architecture for the internet protocol. RFC 2401, November 1998.

[MMFR96] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. TCP Selective Acknowledgement options. RFC 2018, April 1996.

[PAD+99] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, and B. Volz. Known TCP implementation problems. RFC 2525, March 1999.

[RHE99] Reza Rejaie, Mark Handley, and Deborah Estrin. RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the Internet. In *INFOCOM '99*, March 1999.

[Sch96] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, 2nd edition, 1996.

[She94] Scott Shenker. Making greed work in networks: A game-theoretic analysis of switch service disciplines. In *SIGCOMM '94*, pages 47–57, August 1994.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated*, volume 1. Addison Wesley, 1994.

[Vas] Fyodor Vaskovich. nmap. `http://www.insecure.org/nmap/`.

[VRC98] L. Vivisano, L. Rizzo, and J. Crowcroft. TCP-like congestion control for layered multicast data transfer. In *INFOCOM '98*, April 1998.

[ZDE+93] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Network*, pages 8–18, September 1993.