Causing Remote Hosts to Renege

Nasif Ekiz
F5 Networks
Seattle, WA
n.ekiz@f5.com

Paul D. Amer
Computer and Information Sciences Department
University of Delaware
Newark, DE
amer@udel.edu

Fan Yang
Computer and Information Sciences Department
University of Delaware
Newark, DE
yangfan@udel.edu

***Abstract*— Reneging occurs when a data receiver first selectively acknowledges data, and later discards that data from its receiver buffer before delivery to the receiving application. The consequences of reneging on operating systems and active transport connections are unknown. This paper investigates if reneging helps an operating system to resume its operation, and if a reneged TCP connection can complete a data transfer. To document consequences of reneging, we inspect operating systems and TCP connections after reneging occurs. To cause reneging, we present a tool that exhausts system resources by filling TCP receive buffers of a remote host with out-of-order data on multiple TCP connections. FreeBSD, Solaris and Windows were reneged, and their consequences of reneging are reported herein.**

***Keywords*— End-host behavior; protocol measurement; reneging; SACK; Selective Acknowledgment; TCP**

I. MOTIVATION

Transmission Control Protocol (TCP) [13] uses sequence numbers and cumulative acknowledgments (ACKs) to achieve reliable data transfer. Sequence numbers help a TCP data receiver sort and reorder arriving data segments. Data arriving in expected order, i.e., ordered data, are cumulatively ACKed to the data sender. With receipt of an ACK, the data sender assumes the data receiver accepts responsibility of delivering the ACKed data to the receiving application, and deletes all the ACKed data from its send buffer, potentially even before that data are delivered to the receiving application.

The Selective Acknowledgment Option (SACK), specified in RFC2018 [11], extends TCP's cumulative ACK mechanism by allowing a data receiver to ack arrived *out-of-order data* to the data sender. The intent is that SACKed data do not need to be retransmitted during loss recovery. Prior research [1, 3, 8] shows that SACK improves TCP throughput when multiple losses occur within the same window.

Data reneging (or simply reneging) occurs when a data receiver first SACKs data, and later discards that data from its receiver buffer prior to delivering it to the receiving application or socket buffer. TCP is designed to tolerate reneging. Specifically RFC2018 states: "*The SACK option is advisory*" and "*the data receiver is permitted to later discard data which have been reported in a SACK option*". Reneging might happen, for example, when an operating system needs to recapture previously allocated memory, say to avoid a deadlock or to protect the operating system against a denial-of-service attack (DoS). Reneging is possible in FreeBSD, Linux, Mac OS X, Solaris and Windows.

The consequences of reneging on operating systems and active transport connections are unknown. To understand the consequences of reneging, one must answer the following two questions.

(1) Does reneging help an operating system avoid crashing, thereby resuming normal operation? If yes, we can conclude that reneging is a useful and essential mechanism. If a machine still cannot resume normal operation (i.e., it crashes) after reneging, then why bother implementing reneging? And why bother designing transport protocols to support reneging?

(2) Can an active TCP connection complete a data transfer successfully when some out-of-order data are reneged? In general, a TCP data sender does not have a mechanism to infer reneging. To tolerate reneging, a sender is expected to discard its SACK scoreboard upon a retransmission timeout, and retransmit all bytes starting from the left edge of the window [11]. If a TCP sender does not tolerate reneging properly, reneging may cause a data transfer to fail.

To answer (1) and (2), one should analyze a reneging host and its connections before, during and after reneging. Unfortunately reneging is uncommon, and monitoring connections waiting for reneging to happen is impractical. Instead of waiting for this rare event, a tool to cause reneging on a remote host can be developed to investigate its consequences in a lab-controlled environment. Using such a tool, remote hosts with different operating systems can be analyzed in detail.

Our preliminary investigation of various TCP implementations revealed that reneging in general is scheduled to happen (a) when system resources such as main memory/network buffers become scarce, or (b) when out-of-order data sit in a receive buffer for long time without being delivered to a receiving application [7]. To cause reneging, a tool can exhaust system resources by filling TCP receive buffers of a remote host only with out-of-order data thus simulating both (a) and (b).

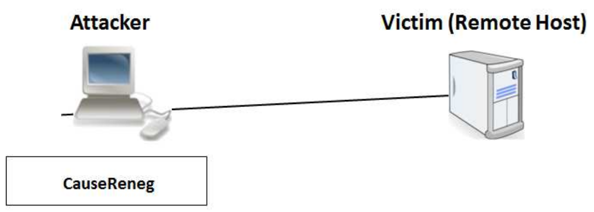

Fig. 1. Causing a remote host to renege

Our tool to cause a remote host to renege is called CauseReneg. Fig. 1 depicts CauseReneg's architecture. In this work, we present CauseReneg, its application on FreeBSD, Solaris, and Windows, and consequences of reneging on those operating systems.

The rest of this paper is organized as follows. Section II details our tool to cause remote hosts to renege. The consequences of reneging on various operating systems are presented in Section III. Finally, Section IV concludes our work.

II. A Tool to Cause Reneging

CauseReneg is hostile to the victim's operating system, and falls into the category of a denial-of-service attack (DoS) tool. CauseReneg exhausts a victim's resources by filling its TCP receive buffer almost fully with out-of-order data. A victim's TCP allocates main memory and network buffers to store that out-of-order data in a receive buffer (a.k.a., reassembly queue). Since out-of-order data cannot be delivered to the receiving application, resources are utilized for the entire time the out-of-order data sit in the receive buffer.

To further exhaust main memory and network buffers, CauseReneg establishes *n* parallel TCP connections to a victim. As *n* increases, a victim uses more and more main memory and network buffers. If all goes as expected, eventually reneging occurs and the main memory used for out-of-order data is reclaimed back to the victim's operating system.

We implemented CauseReneg using the TCP Behavior Inference Tool (TBIT) [12] which can be downloaded from [4]. CauseReneg initiates *n* CauseReneging TBIT tests to establish *n* TCP connections in parallel with a victim. The number of parallel TCP connections (*n*) used by CauseReneg tool is variable and changes with a victim's main memory, available network buffers, and operating system. The CauseReneging TBIT test is shown in Fig. 2 and operates as follows:

CauseReneging

1. TBIT establishes a connection with SACK-Permitted option and Initial Sequence Number (ISN) 10000
2. Victim replies with SACK-Permitted option
3. TBIT sends in-order segment (10001-10006)
4. Victim acks the in-order data with ACK (10006)
5. TBIT skips sending 1455 bytes (10006-11461) and starts sending *m* consecutive out-of-order segments each 1460 bytes to exhaust main memory
6. Victim acks the out-of-order data with SACKs
7. TBIT sends a 10 byte out-of-order segment after *x* seconds
8. Victim acks the out-of-order data with a SACK
9. TBIT sends *m+1* in-order data segments to complete the data transfer
10. Victim acks the in-order data with ACKs/SACKs
11. TBIT sends three RSTs to abort the connection

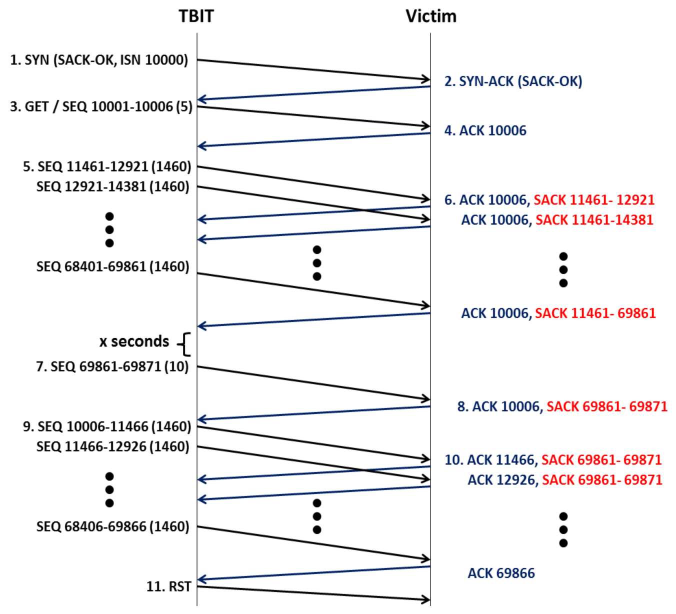
TBIT
Victim
1. SYN (SACK-OK, ISN 10000)
2. SYN-ACK (SACK-OK)
3. GET / SEQ 10001-10006 (5)
4. ACK 10006
5. SEQ 11461-12921 (1460)
SEQ 12921-14381 (1460)
6. ACK 10006, SACK 11461- 12921
ACK 10006, SACK 11461-14381
SEQ 68401-69861 (1460)
ACK 10006, SACK 11461- 69861
x seconds
7. SEQ 69861-69871 (10)
8. ACK 10006, SACK 69861- 69871
9. SEQ 10006-11466 (1460)
SEQ 11466-12926 (1460)
10. ACK 11466, SACK 69861- 69871
ACK 12926, SACK 69861- 69871
SEQ 68406-69866 (1460)
ACK 69866
11. RST

Fig. 2. The CauseReneging TBIT test with *m*=40

Now let us explain the CauseReneging TBIT test in detail. First, a TCP connection is established to a victim with 3-way handshake (step #1, #2, and #3) with SACK-Permitted option. A 5 byte in-order data is sent to the victim along with the ACK (step #3) that establishes the connection. Next, the victim's receive buffer is filled with *m* out-of-order segments (step #5) based on the receiver's advertised window (step #2).

As more TCP connections to the victim are established, we expect reneging to happen. Let us assume that reneging happens after *y* seconds. In (step #7), a 10 byte out-of-order data is sent after *x* seconds. The *x* second value (step #7) is set to a value greater than *y* to detect reneging using the response SACK (step #8). If that response SACKs only 10 bytes of out-of-order data as shown in Fig. 2, one can conclude reneging had occurred.

To mimic an RFC2018 conformant SACK implementation, *m+1* in-order segments are retransmitted (step #9) next, assuming a retransmission timeout value of *x* seconds. Recall that a TCP data sender is expected to discard SACK scoreboard at a retransmission timeout and retransmit bytes at the left edge of the window as specified in RFC2018. If reneging had happened, ACKs (step #10) will increase steadily after each in-order retransmission (step #9) as shown in Fig. 2. If reneging did not happen, the first ACK (step #10) will acknowledge all out-of-order data at once.

CauseReneg is a generic tool that can cause reneging on victims running on various operating systems. Minimal changes needed are to set the *m* value (step #5, #9) and *x* value (step #7) in the CauseReneging test, and the number of parallel TCP connections *n* that change dynamically from victim to victim. These values are determined by the victim's operating system, available main memory, and network buffers.

To establish the TCP connections, CauseReneg needs the victim to have an accessible port. A victim's server socket must be listening on the port to accept incoming TCP connections. Today, the majority of ports are blocked by firewalls for security. Web servers on the contrary remain accessible. Therefore CauseReneg was designed to attack a victim which deploys a web server (step #3 in Fig. 2 sends the first 5 bytes of a HTTP GET request).

To cause reneging, we installed Apache 2.2 in all potential victims. By default, Apache supports at most 256 TCP simultaneous connections. A busy web server with thousands of TCP connections is a stronger candidate to renege. To simulate a busy web server, we increased the default limit to 2000 simultaneous connections which turned out to be enough to cause all victims to renege.

Fig. 3 presents our experimental design. CauseReneg can attack victims regardless of their operating systems when a web server (e.g., Apache) is running. A packet capture utility, tcpdump [14], records TCP traffic between CauseReneg and a victim for later offline analysis. By analyzing the recorded TCP traffic, we can detect reneging instances as detailed in [6].

Reneging is possible in FreeBSD, Linux, Mac OS X, Solaris and Windows. In Mac OS X, reneging is not possible by default, but can be turned on by a system administrator by enabling the sysctl *net.inet.tcp.do_tcpdrain*. We attempted to cause reneging on the following operating systems in which reneging is possible by default: FreeBSD 8.1, Linux 2.6.31, Solaris 11, Windows Vista and Windows 7. We excluded Mac OS X since reneging is not possible by default.

III. Results

Four out of the five operating systems (victims) tested were successfully reneged. We failed to cause a Linux 2.6.31 victim to renege. Linux implements dynamic right-sizing (DRS) where the receiver's window (rwnd) dynamically changes based on the receiver's estimate of the sender's congestion window (cwnd) [9]. A data receiver increases rwnd when in-order data are received meaning the cwnd is increased. The initial advertised rwnd in Linux is 5840 bytes. CauseReneging sends only 5 bytes in-order data (step #3). Therefore, rwnd is not increased and limits CauseReneg to send 4380 (5840 – 1460) bytes of out-of-order data to a Linux victim.

In Linux, reneging is expected when the memory allocated for receive buffer exceeds the memory limit available to the receive buffer. The minimum size of the receive buffer is specified with *net.ipv4.tcp_rmem* sysctl and is initialized to 4096 bytes. Apparently, sending 4380 bytes of out-of-order data was not enough to exceed the memory limit available to the receive buffer. Thus, DRS prohibited CauseReneg to send more out-of-order data to trigger reneging. As a result, CauseReneg was unable to cause reneging in Linux.

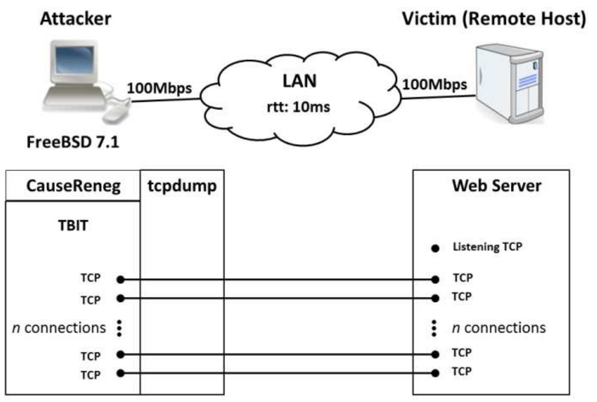

Fig. 3. Experimental design

The consequences of causing FreeBSD, Solaris, and Windows to renege are presented in subsections A, B and C, respectively.

A. *Causing a FreeBSD Host to Renege*

A victim running FreeBSD 8.1, and Apache 2.2 was attacked. The victim had ~500MB physical memory. Two attacks were performed. The first crashed the system, the second caused reneging.

In FreeBSD, a TCP reassembly queue is bounded by at most *net.inet.tcp.reass.maxqlen* ("Maximum number of TCP segments per individual Reassembly queue") out-of-order segments. The default value is 48 segments. CauseReneg can fill a reassembly queue almost fully with out-of-order data since the victim's advertised TCP receive window of 65535 bytes is less than the reassembly queue limit.

Another sysctl, *net.inet.tcp.reass.maxsegments* ("Global maximum number of TCP segments in Reassembly queue"), defines a global limit for all segments in all connections' reassembly queues. FreeBSD assigns 1/16th of total mbuf clusters to *net.inet.tcp.reass.maxsegments*. After that limit is reached, arriving out-of-order segments are dropped. That limit for the attacked victim was 1060. The *net.inet.tcp.reass.overflows* ("Global number of TCP Segment Reassembly Queue Overflows") sysctl reports the total number of dropped out-of-order segments. The *net.inet.tcp.reass.cursegments* ("Global number of TCP Segments currently in Reassembly Queue") sysctl reports the total number of segments in all reassembly queues. Fig. 4 will contain an example output.

In FreeBSD, reneging happens if the page replacement daemon (*vm_pageout*) invokes the *vm_pageout_scan*() function. When the available main memory goes low, and hard-coded or tunable paging thresholds are exceeded, *vm_pageout_scan*() is invoked to scan main memory to free some pages. If the memory shortage is severe enough, the largest process is also killed [2].

Attacks. In the first attack (A), CauseReneg established a variable number of parallel TCP connections. The goal was to exhaust the victim's main memory as much as possible and trigger reneging. Table I presents the memory statistics when *n* parallel TCP connections were established. Each TCP connection exhausts ~2.8MB of main memory. While we expected reneging to happen with increased memory usage, instead the victim crashed! More precisely, when the number of parallel connections exceeded 1241, the victim crashed with the following panic messages: (a) "Approaching the limit on PV entries, consider increasing either the *vm.pmap.shpgperproc* ("Page share factor per proc") or the *vm.pmap.pv_entry_max* ("Max number of PV entries") tunable", and (b) "panic: *get_pv_entry*: increase vm.pmap.shpgperproc". The panic messages are related to mapping of physical/virtual addresses of pages.

To track the number of connections causing the victim crash, CauseReneg attacked the victim with the following configuration: *n*=1300, *m*=1, *x*=240 seconds. Each TCP connection sent only 1 out-of-order segment to the victim. Fig. 4 shows the statistics for TCP reassembly queue size when 1241 parallel TCP connections were established to the victim just before crashing (*net.inet.tcp.reass.cursegments:*1059 (line 4) + *net.inet.tcp.reass.overflows*: 182 (line 2) = 1241).

TABLE I. MEMORY USAGE FOR N PARALLEL TCP CONNECTIONS

n Parallel TCP Connections	Memory Usage
1	3MB
2	6MB
4	11MB
8	22MB
200	558MB
400	1127MB
800	2267MB
1241	3418MB

In the second attack (B), to cause the page replacement daemon to call the *vm_pageout_scan*() function, a user process that consumes a specified amount of main memory is executed along with CauseReneg. If the memory shortage is severe enough due to the user process' excessive memory allocation, and the victim goes low on main memory, the page replacement daemon is expected to kill the process using the largest memory (in that case, the user process) and renege.

The second attack was performed for two cases: (B1) reneging was enabled (*net.inet.tcp.do_tcpdrain*=1), and (B2) reneging was disabled (*net.inet.tcp.do_tcpdrain*=0) at the victim.

Results. For attack (A) although memory consumption was high (3533MB: 3418MB for the TCP connections + 115MB for the kernel), reneging did not occur. The reason is that the paging thresholds were not exceeded. If reneging happened, FreeBSD would reclaim ~3MB of main memory (all available space for out-of-order data). Since each TCP connection established consumes ~2.8MB, reclaimed memory would be consumed for only one new TCP connection. Eventually, the machine crashes. We conclude that reneging does not benefit FreeBSD for such an attack.

```
1 [nekiz@muscat ~]$ sysctl -a | grep tcp.reass
2 net.inet.tcp.reass.overflows: 182
3 net.inet.tcp.reass.maxqlen: 48
4 net.inet.tcp.reass.cursegments: 1059
5 net.inet.tcp.reass.maxsegments: 1060
```

Fig. 4. Statistics for TCP reassembly queue size (attack A)

For attack (B1), CauseReneg attacked the victim with *n*=20, *m*=40, *x*=180 seconds. When 20 parallel connections were established, 800 out-of-order segments were in the TCP receive buffers (*net.inet.tcp.reass.cursegments:* 800). Next, the user process was run to allocate 2GB of main memory. FreeBSD allocated ~1.5GB of main memory to the user process before the user process was killed by the page replacement daemon. At this point, reneging was expected and *net.inet.tcp.reass.cursegments:* 0 confirmed that reneging did occur.

Attack (B2) was performed with the same steps as attack (B1). When the user process was terminated by the page replacement daemon, the out-of-order segments were still in the TCP receive buffers (*net.inet.tcp.reass.cursegments:* 800) confirming that reneging did not happen.

Both attacks (B1) and (B2) also were analyzed using the RenegDetect tool [6] which identifies reneging and non-reneging connections.

The FreeBSD victim was reneged with the attack (B1). Now, we answer the following questions to gain insight to the consequences of reneging: (1) Does reneging help an operating system to resume its operation? (2) Can a reneged TCP connection complete its data transfer?

(1) After attack (B1), the FreeBSD victim kept running normally. As stated before, only ~3MB of main memory (the maximum amount possible for the victim) used for the network buffers was reclaimed back to the operating system. Since the memory shortage caused by the attack was severe, the largest process (~1.5GB) was killed. We believe the amount of main memory used for network buffers is negligible compared to the process using the most memory. We argue that the current handling of reneging is not useful, and reneging should be turned off by default in FreeBSD as in Mac OS X.

To answer (2), we needed to test if the TCP data senders tolerate reneging properly as specified in RFC2018. Recall that a TCP sender needs to discard its SACK scoreboard at a retransmission timeout, and start sending bytes at the left edge of the window. Otherwise, reneging may cause a data transfer to stall and/or fail.

FreeBSD employs a global reneging strategy that all TCP connections with out-of-order data are reneged simultaneously. If TCP connections with out-of-order data from various TCP data senders were established to the FreeBSD victim before the attack (B1), those TCP connections would renege too. To test if RFC2018 conformant tolerating reneging is implemented, a 5MB file was transferred using secure shell (ssh) to the FreeBSD victim from various operating systems: FreeBSD 8.0, Linux 2.6.24, Mac OS X 10.8.0, NetBSD 5.0.2, OpenBSD 4.8, OpenSolaris 2009, Solaris 11, Windows XP, Windows Vista, and Windows 7.

To create out-of-order data for those transfers, Dummynet [5] was configured on the FreeBSD victim to drop 15-20% of the TCP PDUs. The traffic between a TCP data sender and the FreeBSD victim was recorded for reneging analysis. Once a data transfer started, the FreeBSD victim was reneged using the attack (B1). Then we observed if the file transfer experiencing reneging would be completed. In all data transfers, reneging was detected by analyzing the recorded traffic using RenegDetect, and we confirmed that all of the TCP data senders completed the data transfer successfully. The conclusion is that RFC2018 conformance is implemented in all TCP stacks tested.

B. *Causing a Solaris Host to Renege*

In Solaris, if out-of-order data sit in the TCP reassembly queue for at least 100 seconds (i.e., the default reassembly timer timeout value), a receiver would renege and purge the entire reassembly queue. Reneging, in such case, protects the operating system against DoS attacks.

Attack. Reneging in Solaris is expected to happen 100 seconds after the arrival of out-of-order data (CauseReneging TBIT test step #5). To force the reassembly queue timer to expire, we set $x > 100$ seconds. The number of parallel connections (n) and out-of-order segments (m) can be set arbitrarily since reneging in Solaris only depends on x. CauseReneg attacked the victim with n=20, m=40, x=180 seconds. The value of m was set to 40 purposefully to explain reneging using Fig. 2. With this configuration, reneging was expected to happen before the 10 byte out-of-order data were sent (step #7).

Results. A Solaris 11 victim with 1024MB physical memory and running Apache 2.2 was tested. As expected, reneging happened when the TCP reassembly queue timer expired after 100 seconds for the out-of-order data sent (step #5) in Fig. 2. A tcpdump output of the 6th parallel connection is shown in Fig. 5. After x=180 seconds, 10 byte out-of-order data (69861-69871) were sent (lines 5, 6). The reply SACK evidenced reneging since only 10 out-of-order bytes were selectively acknowledged (69861-69871) (lines 7, 8). After the in-order received data, the victim's ACKs (step #10) were steadily increased as expected.

We believe reneging in Solaris is used as a mechanism to protect against DoS attacks. In loss recovery, a TCP sender is expected to retransmit a lost segment r times (e.g., *TcpMaxDataRetransmissions* in Windows Server 2003 defines r=5 by default). After r retransmissions, a TCP sender would terminate a TCP connection. The loss recovery period takes 1-2 minutes assuming back to back timeouts, an initial retransmission timeout value (RTO) of 1 second, and r=5. When out-of-order data sit in the reassembly queue for at least 100 seconds (the default reassembly queue timer value) at the Solaris receiver, one can infer that either the TCP sender terminated the connection or the host is under a DoS attack where the out-of-order data intentionally exhaust host's resources. Therefore, cleaning the reassembly queue seems a useful mechanism in both cases.

```
1 21:06:43.572124 IP 128.4.30.32.20005 > 128.4.30.29.80:
2     P 2881168401:2881169861(1460) ack 128403387 win 21900
3 21:06:43.572768 IP 128.4.30.29.80 > 128.4.30.32.20005:
4     . ack 2881110006 win 64240 <nop,nop,sack 1 {2881111461:2881169861}>
5 21:09:43.579360 IP 128.4.30.32.20005 > 128.4.30.29.80:
6     P 2881169861:2881169871(10) ack 128403387 win 21900
7 21:09:43.579739 IP 128.4.30.29.80 > 128.4.30.32.20005:
8     . ack 2881110006 win 64240 <nop,nop,sack 1 {2881169861:2881169871}>
```

Fig. 5. Tcpdump output of a TCP connection from reneging attack on Solaris 11

Instead of reneging out-of-order data, we believe a better option would be to RESET the connection when reneging is caused by either a terminated TCP connection (due to loss recovery) or a DoS attack. With that change, all the resources used for the TCP connection would be released, and better utilized.

C. *Causing Windows Hosts to Renege*

Dave MacDonald, author of Microsoft Windows 2000 TCP/IP Implementation Details [10], stated that Vista and its successors implement reneging as a protection mechanism against DoS attacks. Reneging happens when the memory consumption of total TCP reassembly data in relation to the global memory limits is significant.

To investigate the consequences of reneging, CauseReneg attacked Windows victims (Vista and 7) by increasing the number of parallel connections to make the memory consumption of the total reassembly data so significant that reneging is triggered. The Vista and Windows 7 victims had 2GB and 1GB physical memory, respectively. Both victims deployed Apache 2.2.

Attacks. The initial advertised window (rwnd) in both Vista and Windows 7 is 64240 bytes. This value corresponds to 44 * 1460 byte TCP PDUs. Based on the initial rwnd, the m value, the number of out-of-order segments, in the CauseReneging test (step #5) is set to 43 to fill each reassembly queue almost fully with out-of-order data. In the attacks, the number of parallel connections established to the victim (n) and the x seconds (step #7) values in the CauseReneging test were variable.

Results. First, the Vista victim was attacked by CauseReneg using m=43, x=200. Table II presents the results of the attacks. When n=100 or 200 parallel connections were established , reneging did not happen. When n=300 parallel connections were established, only the last 33 connections reneged. When the n=400, a similar behavior happened. The first 267 connections did not renege but the last 133 connections did. This behavior implies that the memory consumption of total reassembly data in relation to the global memory limit is considered significant in Vista when the out-of-order data in the reassembly queue is at least ~16MB (= 267 (parallel connections) * 43 (out-of-order segments) * 1460 bytes).

To verify that the global memory limit for reneging is ~16MB, another attack was performed with the configuration: n=600, m=20, x=200 seconds. With this configuration, roughly half of the rwnd was filled with out-of-order data. The observed behavior was consistent: only the last 25 of 600 connections reneged, and the memory allocated to out-of-order data before reneging happened was again ~16MB (= 575 (parallel connections) * 20 (out-of-order segments) * 1460 bytes).

TABLE II. CAUSERENEG ATTACK ON VISTA VICTIM WITH VARIABLE NUMBER OF PARALLEL CONNECTIONS

***n* parallel TCP connections**	**Reneging**
100	No
200	No
300	Yes (33 connections renege)
400	Yes (133 connections renege)

Next, the Windows 7 victim was attacked by CauseReneg using *m*=43, *x*=200. Table III presents the results of the attacks. When *n*=100 parallel connections were established, reneging did not happen. When *n*=200 parallel connections were established, the first 133 connections did not renege but the last 67 connections did. When *n*=300 parallel connections were established, the first 133 connections did not renege but the last 167 connections did. This behavior implies that the memory limit for the reassembly queue for the Windows 7 victim was ~8MB (= 133 (parallel connections) * 43 (out-of-order segments) * 1460 bytes). Recall that the Vista victim had a physical memory of 2GB whereas the Windows 7 victim's memory was 1GB. The memory limit used for the reassembly data to trigger reneging in both systems was ~0.78% of the physical memory, and seems to scale with the physical memory.

TABLE III. CAUSERENEG ATTACK ON WINDOWS 7 VICTIM WITH VARIABLE NUMBER OF PARALLEL CONNECTIONS

***n* parallel TCP connections**	**Reneging**
100	No
200	Yes (67 connections renege)
300	Yes (167 connections renege)

In conclusion, Windows Vista and 7 support reneging as a protection mechanism against DoS attacks, renege when the memory threshold for reassembly data is reached, and resume normal operation after reneging.

IV. CONCLUSIONS

CauseReneg was used to attack FreeBSD, Solaris, and Windows. Two attacks were performed on a FreeBSD victim. Initially, it was thought that an operating system starving for main memory would eventually crash. Our first attack (A) was such an example: the victim crashed and reneging did not help the operating system resume normal operation. In the second attack, the page replacement daemon invoked drain routines, the victim reneged, and the OS resumed normal operation. On the other hand, attack (B2), where reneging was disabled for the second attack, demonstrated that FreeBSD could resume normal operation again without reneging. Thus, reneging did not benefit FreeBSD in either attack.

The maximum amount of memory that can be allocated to a victim's reassembly queues by reneging is limited to ~3MB (0.6% of the physical memory). That amount of memory seems negligible compared to the process using the most memory. We believe reneging alone does not help an OS resume normal operation, and the reassembly queues' memory was wastefully purged. Therefore, we argue that reneging support should be turned off by default in FreeBSD as it is in Mac OS X.

Both Solaris 11 and Windows (Vista and 7), use reneging as a protection mechanism against DoS attacks instead of using reneging to manage memory pressure situations. One key difference is that Solaris uses a reassembly queue timer to renege whereas Windows uses a memory threshold for the out-of-order data for the same purpose.

In Solaris, when out-of-order data sit in the reassembly queue for at least 100 seconds, reneging is triggered. It can be inferred that the connection is either terminated due to loss recovery or exhausts resources intentionally (a DoS attack). In both cases, instead of reneging, we contend that terminating the connection with RESETs would be a better option since RESETing releases all of the allocated resources.

In Windows, reneging happens when the memory allocated for out-of-order data exceeds the memory threshold available for the reassembly data. The current reneging implementation in Windows has a potential problem. The out-of-order data that cause reaching the threshold are not reneged. Instead, it is the out-of-order data received afterwards that are reneged. Were an attacker to find out the memory threshold (as we did in Section III.C) and only send that amount of out-of-order data, all future connections experiencing losses and receiving out-of-order data afterwards would renege. A TCP data sender would not retransmit SACKed data until a retransmission timeout (RTO) [11]. In such a case, losses would be recovered with RTOs resulting in increased transfer times (lower throughput). The quality of service for legitimate users would be reduced, i.e., data transfer times would increase. We introduce that type of an attack as a *reduction of service* (RoS) attack. We believe that an RoS attack would be harder to detect compared to a DoS attack since the service provided in not interrupted but only slowed down.

When we compare reneging in Solaris vs. Windows, Solaris's approach seems to be a better protection mechanism: only the DoS connections are penalized.

In conclusion, when an OS is starving for memory, reneging alone does not help the system resume normal operation. Therefore, we argue that reneging support should be turned off for systems employing that type of reneging. Reneging in Solaris and Windows protects the system against DoS attacks. We argue that type of protection is essential to operating systems, but we believe that a better approach would be to RESET the connection under the attack instead of reneging.

REFERENCES

[1] M. Allman, C. Hayes, H. Kruse, and S. Ostermann, "TCP performance over satellite links," in Proceedings of the 5th International Conference on Telecommunication Systems, 1997, pp. 456–469.

[2] M. Bruning, "A comparison of Solaris, Linux, and FreeBSD Kernels", http://hub.opensolaris.org/bin/view/Community+Group+advocacy/solaris-linux-freebsd

[3] R. Bruyeron, B. Hemon, and L. Zhang, “Experimentations with TCP selective acknowledgment,” ACM SIGCOMM Computer Communication Review, vol. 28, no. 2, p. 54, 1998.

[4] CauseReneg, http:// http://pel.cis.udel.edu/CauseReneg.tar

[5] Dummynet, http://info.iet.unipi.it/~luigi/dummynet/

[6] N. Ekiz and P. D. Amer, “A model for detecting transport layer data reneging,” in The 8th International Workshop on Protocols for Future, Large-Scale & Diverse Network Transports, 2010.

[7] N. Ekiz, “Transport layer reneging”, PhD Dissertation, Computer and Information Science Department, University of Delaware, 2012.

[8] K. Fall and S. Floyd, “Simulation-based comparisons of Tahoe, Reno and SACK TCP,” ACM SIGCOMM Computer Communication Review, vol. 26, no. 3, pp. 5–21, Jul. 1996.

[9] M. Fisk and W. Feng, “Dynamic right-sizing: TCP flow-control adaptation,” in Supercomputing (SC01), 2001, pp. 1–3.

[10] D. MacDonald and W. Barkley, “Microsoft Windows 2000 TCP/IP implementation details”, http://microsoft.technet.microsoft.com/en-us/library/bb726981.aspx

[11] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, “TCP selective acknowledgment options”, RFC2018, 1996.

[12] J. Pahdye and S. Floyd, “On inferring TCP behavior,” ACM SIGCOMM Computer Communication Review, vol. 31, no. 4, pp. 287–298, Oct. 2001.

[13] J. Postel, “Transmission control protocol”, RFC793, 1981.

[14] Tcpdump, www.tcpdump.org