

Performance tradeoffs of TCP Selective Acknowledgment

Does the SACK optimization also present a denial-of-service opportunity?

Skill Level: Advanced

[Patrick McManus \(mcmanus@ducksong.com\)](mailto:mcmanus@ducksong.com)
Independent consultant

31 Mar 2008

Selective acknowledgment (SACK) is an optional feature of TCP that is necessary to effectively use all of the available bandwidth of some networks. While SACK is good for throughput, processing this type of acknowledgment has proven to be CPU intensive for the TCP sender. This weakness can be exploited by a malicious peer even under commodity network conditions. This article presents experimental measurements that characterize the extent of the problem within the Linux® TCP stack. SACK is enabled by default on most distributions.

Mixed in with the usual Linux® development chatter on the Internet over the last few months has been a significant discussion of Linux's TCP SACK (Selective Acknowledgment) implementation. These comments have generally focused on the performance of the TCP stack when processing certain SACK events, and some people have hinted at the presence of a security exposure.

I was intrigued by the discussion, but it seemed to lack hard data. What specific conditions were they talking about? Is this a minor performance nit, or is it an outright server denial-of-service (DoS) opportunity?

I collected a few quotes on the subject (see [Resources](#) for links to the sources of these):

David Miller: "This is a problem that basically every single TCP stack out there right now is vulnerable to, lots of CPU processing for

invalid or maliciously created SACK blocks."

Ilpo Jarvinen [1]: "However, as long as there is dependency[sic] to skb's `fack_count` in `sacktag`, there seems to be some room for CPU processing attacks even with RB-tree case because walk becomes necessary on the slow path."

NC State University: "We show the efficiency of SACK processing in this experiment. As we have seen a lot of cases that TCP didn't work well with large window drop especially with large buffer."

CHC IT: "And finally a warning for both 2.4 and 2.6: for very large BDP paths where the TCP window is > 20 MB, you are likely to hit the Linux SACK implementation problem. If Linux has too many packets in flight when it gets a SACK event, it takes too long to locate [sic] the SACKed packet, and you get a TCP timeout and CWND goes back to 1 packet."

This article looks at the SACK implementation and its performance under non-ideal conditions starting with Linux 2.6.22, which is currently used as a common distribution kernel on Ubuntu 7.10. That kernel is several months old now, and since its release the developers haven't just been talking about the issue, they have been writing code for it. The current development kernel, 2.6.25, contains a series of patches from Ilpo Jarvinen that deal with SACK performance. I finish up the article by looking at how that code may change things and also take a quick look at some more changes under consideration farther down the road.

A SACK primer

SACK is defined by RFCs 2018, 2883, and 3517 (see [Resources](#) for links to these RFCs). Plain TCP (in other words, non-SACK) acknowledgments are strictly cumulative—an acknowledgment of N means that byte N and all previous bytes have been received. The problem SACK is meant to address is this "all or nothing" nature of the plain cumulative acknowledgment.

For instance, even if packet 2 (in a sequence 0 to 9, say) is the only packet lost during a transfer, the receiver can issue a plain ACK only for packet 1, because that is the highest packet it received without a gap. A SACK receiver, on the other hand, can issue an ACK for 1 plus a SACK option for packets 3 through 9. This extra information helps the sender determine that the losses are fairly minimal and that it only needs to retransmit a little bit of data. Without this extra information, it would need to retransmit much more data and slow down its sending rate to accommodate what looks like a high-loss network.

SACK is especially important to effectively utilize all available bandwidth on

high-delay connections. Because of the high delay, there are often large numbers of packets "in flight" waiting for acknowledgment at any given time. In Linux, these packets sit in the retransmission queue until they have been acknowledged and are no longer needed. The packets are in sequence-number order but are not indexed in any way. When a received SACK option needs to be processed, the TCP stack must find the packet(s) in the retransmission queue that the SACK applies to. The larger the retransmission queue, the harder it is to find the data of interest.

Up to four SACK options may be present in each packet.

The attack scenario

The general exposure stems from the fact that the SACK option receiver can be made to do an arbitrary amount of work based on the reception of a single packet. This N:1 ratio allows SACK senders to overwhelm receivers on much more powerful platforms.

The specific attack scenario for a Linux SACK processor requires a retransmission queue that already holds large numbers of packets. The attacker then sends a packet full of SACK options designed to force the other host to scan that entire queue to process each option. An option that refers to a packet at the end of the queue may require a full list traversal by the receiving TCP stack in order to identify which packet the option refers to. An attacking client can obviously send any SACK option it likes, but less obviously, it can also easily control the size of the peer's retransmission queue. The creator of the SACK option is able to establish the amount of work the option receiver will need to perform for each option received, because it also controls the sizes of the queue.

The number of packets in the retransmission queue is fundamentally driven by the bandwidth delay product (BDP) between the two hosts. Bandwidth of a path is capped by the physical properties of the network—an attacker cannot easily increase bandwidth without acquiring more infrastructure. The attacker can, however, arbitrarily add delay by stalling each acknowledgment a little while before transmission. In order to effectively utilize the bandwidth of this high-delay connection, the server needs to have enough packets in flight so that their aggregate transmission time equals the time it takes to get the delayed acknowledgment back. If it does not do this, there are periods where the network is not moving any packets, and therefore its bandwidth is not fully utilized. High-delay paths require lots of packets in flight to use the network efficiently, and TCP senders aggressively try to fill this window within the limits of congestion control standards.

Delaying acknowledgments effectively increases the size of the retransmission queue, a prerequisite for the attack. For example, filling a relatively slow 10 MByte/sec connection with 1750ms of delay generates a window of over 12,000 packets. Faster connections result in even bigger windows, but part of the

vulnerability stems from the fact that this approach is viable on standard home broadband connections just by introducing large delays.

The client selects the value of its SACK options whenever transmitting a delayed ACK. The sender adds SACK options referring to the data in the packets most recently arrived (that is, those with the highest sequence numbers seen). In this scenario, this data is just starting to have its own ACK delayed, but it can be SACKed now, which causes maximum distance in the retransmission queue between the packets being ACKed and SACKed.

This particular attack scenario (the focus of this article) is known as a *find-first* attack, because it forces the TCP stack to spend excessive amounts of time finding the first byte referenced by the SACK option.

Measuring the attack scenario

Kode Vicious: "Take a Freaking Measurement!"

With the background behind us, let's find out whether this is a serious issue or not. Is this an area for micro-optimization, a full-fledged crisis, or something in between? To figure that out, we need to use real data to characterize the extent of the issue. Task number one is to decide what data to gather and how to measure it.

Experiment setup

The most important item to collect is CPU utilization on the server. Oprofile can do the heavy lifting for this using the standard CLK_UNHALTED counter. It is also interesting to count the number of packets scanned while processing the SACKs, and the average size of the retransmission window. I instrumented the server source code to obtain the packets scanned counter. I also reran the tests without this annotation to make sure I was getting the same results that a standard server would see.

The size of the in-flight window is also interesting data. The average size of the retransmit queue can be calculated from the packets scanned counter if the number of SACK options sent is also tracked at the client. For the non-SACK test cases, I used the typical length of the delayed ACK queue reported by the client as an approximation of the lower bound of the sender's window size.

All of the measurements were taken on a standard Linux server. A custom client was written to drive the experiments and trigger the code paths of interest on the server. The client implemented its own TCP stack and ran on top of the raw socket kernel API. The client was certainly not a complete TCP stack, but it was enough to test the variables of interest.

The client starts the experiment by connecting to the server and sending a simple HTTP request for a 700MB ISO file. It then consumes all the data sent by the server over a normal 100 Mbit/sec network in response. Each packet from the server is acknowledged after a 1750ms delay. The server strives to fill that whole 1750ms window by gradually sending more and more packets at one time before it sees them acknowledged. I observed windows in flight of over 14,000 packets.

The client has a configurable option to add the SACK information relating to the most recently arrived data onto each ACK as it is transmitted. If this option is enabled, the generated acknowledgment contains up to four SACK options. Each option refers to a random 1-byte range of one of the four most recently arrived packets that are currently having their acknowledgments delayed. If fewer than four packets are waiting, then a corresponding number of options are generated.

In order to allow meaningful comparisons, I collected this data from three different vantage points:

1. **Baseline:** In the first test I sought to obtain a baseline measurement. Instead of using a custom client and TCP stack, I used the standard Linux TCP stack and the wget command-line HTTP client.
2. **Custom, no SACK:** The second data point needed to use the custom large window-inducing client, but it did not make use of the SACK option at all. This data point allows the separation of basic effects caused by large windows from those caused by handling malicious SACK options.
3. **Custom, with SACK:** The last data set was gathered using the large window-inducing client while also including four SACK options on every ACK sent.

The server I used was an older, 1.2GHz Athlon XP server.

The measurements

Table 1. Server utilization measurements

Method	ACKs processed	Packets scanned for SACK processing	Elapsed time	CPU utilization	Sampled ticks per ACK	Sampled ticks per KB transferred	Average retransmission queue length
Baseline	252,955	0	1:02	22%	1.72	0.56	5
Custom, no SACK	498,275	0	2:59	9%	1.47	1.03	7,000 - 10,000
Custom, with SACK	534,202	755,368,500	12:47	33%	10.87	8.13	1,414

Misleading data

A few initially surprising data points are easily explained.

Note that the number of ACKs is roughly twice as high for the custom client as compared to the baseline wget. This is because the custom client is designed to aggravate server behavior. As a result, it does not do coalesced acknowledgments. Most TCP stacks coalesce two acknowledgments into one if they are sent back to back without any data. The custom client does have an option to do that, but it was not turned on for this data. A run was made with the coalescing option engaged, and it turned out not to materially impact the results. Read why this is so under [What about a bigger server?](#) below.

The alert reader might also notice that the number of sampled ticks per KB transferred is 16 times as great for the code that triggers the SACK path as compared with the baseline number, but the CPU utilization differential is a more modest 1.5. The key additional piece of data is the elapsed time of the test. The baseline used 22% of the CPU for a little over a minute, where the SACK client used 33% of the CPU over almost 13 minutes. At the end of the day, that's a lot more cycles used by the SACK client to achieve the same amount of data transfer.

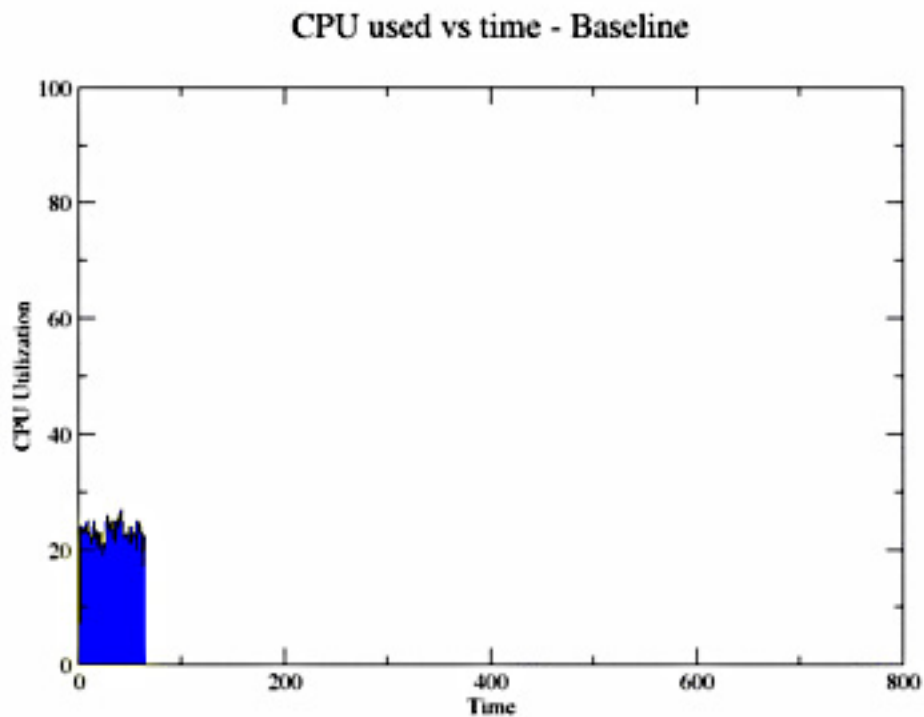
At first glance, these measurements do not appear to be such a bad story after all. Although CPU use is up to 33% during the attack, it is not completely exhausted. Exhausting the CPU would prevent other work from getting done and constitute a denial of service.

Unfortunately, digging just a touch deeper raises some concerns. The total transfer time went way up: from just 1 minute at the baseline to almost 13 minutes under the full attack scenario. Additionally, the increased CPU utilization rate is sustained over that full 13 minutes, as compared to the original 1 minute. In aggregate, that's a lot more cycles expended just to accomplish the same goal. You can see this easily by comparing the sampled ticks per KB transferred number between the three data points.

Digging even deeper shows the 33% CPU utilization number to be misleading. The 13 minutes consist of a repeating cycle of bursts several seconds in duration where the entire server is occupied at 100% utilization. The bursts are followed by lulls in CPU use, and then the cycle repeats again. The overall result is an average 33% utilization, but there are prolonged periods where the CPU is completely monopolized by TCP processing triggered by a remote host.

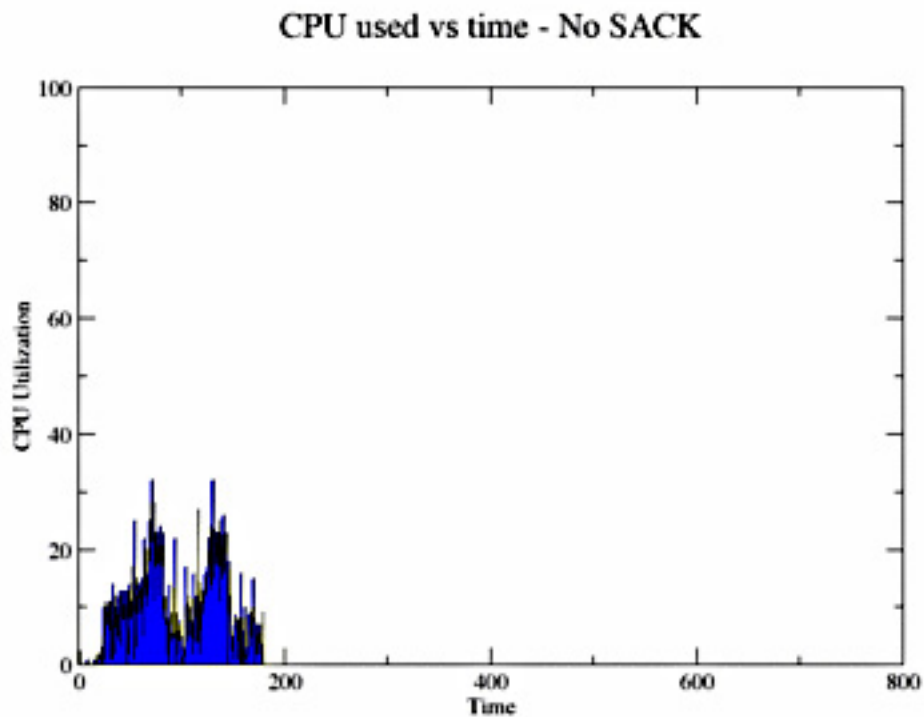
Consider these graphs of CPU utilization vs. time for all three scenarios:

Figure 1. Baseline using wget and no SACK



The baseline measurement is nice and efficient. The transfer completes quickly and fills the available bandwidth without using more than 25% of the CPU at any given time. This proves this modest server is more than capable of filling the network under the right circumstances.

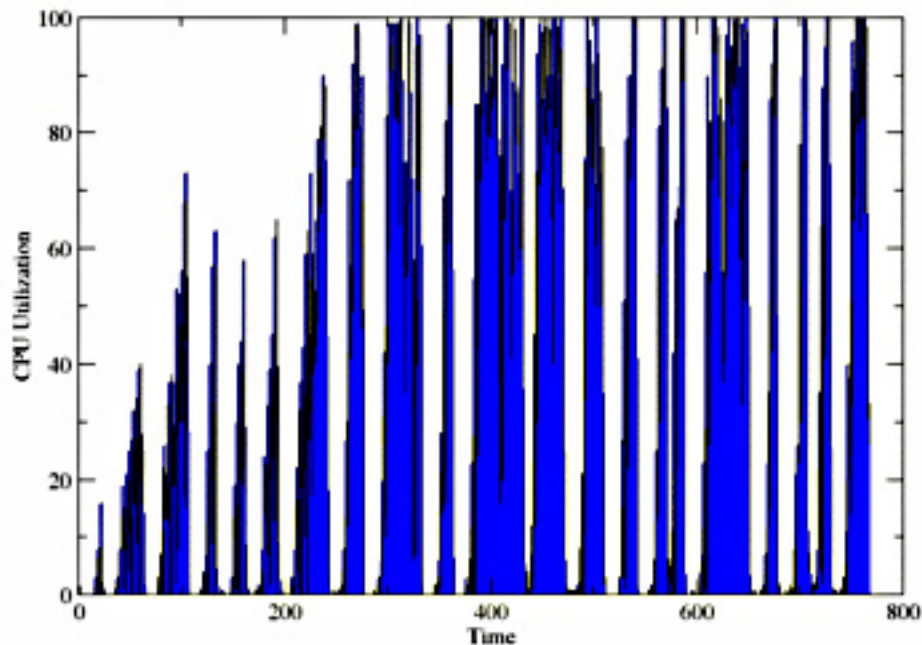
Figure 2. Large-window custom client with no SACK involved



The custom client, when not using SACK, also generates a reasonable utilization graph. The added complication of managing the large windows and their inevitable losses leads to some extra CPU being used, but at no time is the server incapable of keeping up.

Figure 3. Large-window custom client with SACK

CPU used vs time - SACK



You cannot help but be shocked at how much extra blue ink is spilled on this plot of the SACK enabled download. While overall utilization is an average 33%, the graph clearly illustrates recurring bursts where the server is fully swamped.

The graph looks to be repeatedly plotting $y=x^2$ until $y=100\%$. This is the manifestation of each SACK option needing to scan the entire retransmission queue in order to find the data it refers to. When the congestion window on the sender is growing, it effectively doubles the number of packets in flight during the time it takes to send a packet and receive the acknowledgment back. This doubled queue needs to be inspected for each SACK option received. Note the astonishing 755 million packet comparisons done in SACK processing for just a 1/2 million ACKs received. This algorithm generates the exponential behavior seen on the plots.

What about a bigger server?

An initial concern about the experiment was that it used a modest 1.2Ghz Athlon as the server testbed. This would seem to underestimate the capacity of the server side.

However, the quadratic behavior seen here is the kind of trend that cannot be helped by testing a faster server or implementing optimizations such as coalescing two ACKs into one. Such changes will allow the congestion window to grow beyond the modest 3200 packets limits seen during these measurements, but the hockey-stick growth of the CPU utilization on the server will surely still max out before opening the window all the way to 10,000 or

more that is necessary to fill this modest Fast Ethernet network.

The final mystery is why this is not totally catastrophic for the server. It would be reasonable to expect the utilization to ramp to 100% and then stay clamped at that level throughout the rest of the transfer. Instead, we see a series of fits and starts. It is not as if the retransmission queue is getting shorter. Or is it?

In practice the retransmission queue does periodically shrink all the way back to zero packets. From that point the process begins anew, congestion windows again expand, and the SACK overhead picks up again. The whole process takes a number of seconds to reach a size where it impacts the server CPU usage again. That explains the cyclic nature of the graph.

The queue appears to shrink because the intense processing burden at peak causes a TCP timeout-based recovery to occur. The stack reacts to the timeout by resetting the congestion window back to an empty state. This renewed small window hinders transfer speed, but it also keeps the server alive and able to attend to some other work while the window grows into dangerous territory again.

Kernel developments

What is going on in development?

The Linux networking team is already working on this code. On Nov 15, 2007, Ilpo Jarvinen posted a significant rework of the SACK processing path. That code was placed into Linus's pre-2.6.25 tree during the merge window on January 28, 2008. The whole series was 10 patches (see [Resources](#) for links to these). I'll focus here on three key patches of interest.

The patches were written to improve SACK performance in the general case. They might help some with attacking scenarios, but they are not thought to be an antidote to the scenarios measured in this article.

Ilpo Jarvinen [2]: "I think that we cannot prevent malicious guys from inventing ways to circumvent the tweak and optimization that may exist for the typical, legitimate use cases."

The first change (see "Abstract tp->highest_sack accessing & point to next skb" in [Resources](#)) is to optimize the caching strategy in the general case where the SACK option contains only information for data at higher sequence numbers than have been previously SACKed. Generally speaking, this means that an already reported hole remains in the outstanding window, but new data is being SACKed at the more recent end of the window. This is the common case for normal operation. The patch optimizes this case by converting the cached reference from a sequence number to

a pointer to the highest packet in the queue that has been SACKed in the past. Using this information, a later patch (see "Rewrite SACK block processing & sack_rcv_cache use" in [Resources](#)) processes SACKs that only address data higher than this cached value by using the cached pointer as the starting point for the list traversal, which saves most of the work of walking the list.

Unfortunately, the malicious test client cannot be optimized by this strategy. A typical ACK from this client does contain a SACK option for data higher than has been seen before, but it also contains sequence references to the immediately prior packets as well. The 2.6.25 implementation will need to walk the retransmission queue from the start in order to find this data.

This later patch contains a reorganization to support a different "skipping" queue traversal algorithm of the queue with future patches. While this does not immediately help the test results discussed here—as skipping is still implemented with the same linear walk—the future changes supported by this will have a substantial impact on attack scenarios.

The comments included with the patches indicate that two major future changes are in the works. The first potential future change would modify the unacknowledged packet list so that it is organized with an index as a red-black tree instead of the current linear list. This will allow a $\log(N)$ look up of packets referenced by SACK options. A change that introduces some sort of index that allows access to random elements of the large retransmission queue is critical to address the find-first attacks against the TCP stack.

The other organizational change addresses a concern not yet explicitly raised here. The index structure will give good performance looking up individual packets, but a SACK option can cover arbitrary byte ranges that include multiple packets. There is nothing to stop a malicious client from sending options that cover almost all of the data in the window. This is different from the find-first attack I've focused on. Indeed, the first packet may be the first packet in the list and thus very easy to find. However, finding the packets of interest quickly is of little help if the whole queue needs to be walked linearly to process the SACK option. The code change would be a reorganization of the current list into two lists, one with data that had been SACKed and one with data that had not been SACKed. This can help considerably by reducing the search space to just that data not previously SACKed. There are some complications regarding a related specification called DSACK (duplicate SACK), but partitioning is the direction being considered.

The last patch of interest (see "non-FACK SACK follows conservative SACK loss recovery" in [Resources](#)) is a change in congestion control semantics in order to take advantage of the SACK rules in RFC 3517. These changes allow the kernel to avoid full timeout-based recovery scenarios under more circumstances. These timeout-based recoveries require shrinking the sending window all the way down and slowly re-growing it over time back to the level supported by the current

bandwidth delay product. The recovery time is responsible for the pauses between activity bursts during the test.

Measuring 2.6.25-pre

With this new code under our belts, it's time to remeasure the data point that used the custom delay-inducing client with SACK options turned on. This time, the test is done against the 2.6.25 development code. The earlier three data points are included in the table for easy reference.

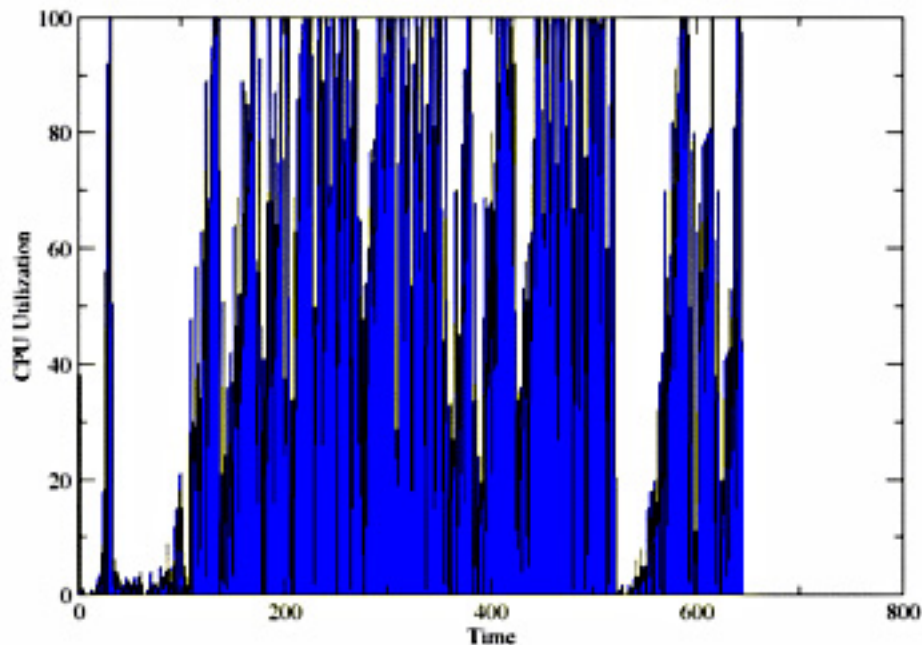
Table 2. Server utilization measurements

Method	ACKs processed	Packets scanned for SACK processing	Elapsed time	CPU utilization	Sampled ticks per ACK	Sampled ticks per KB transferred	Average retransmission queue length
Baseline	252,955	0	1:02	22%	1.72	0.56	5
Custom, no SACK	498,275	0	2:59	9%	1.47	1.03	7,000 - 10,000
Custom, with SACK	534,202	755,368,500	12:47	33%	10.87	8.13	1,414
Custom, with SACK against pre-2.6.25	530,879	2,768,229,470	10:42	49%	13.6	10.07	5,214

This is the graph of CPU usage over time using the custom large-window client with malicious SACK options against a pre-2.6.25 development kernel:

Figure 4. Large-window custom client with SACK against kernel pre-2.6.25

CPU used vs time - SACK on 2.6.25-pre



The CPU graph, which had previously consisted of cyclical fits of starts and stops, is now more consistently saturated over time. Even though the kernel code has been made meaningfully more efficient, the test uses more cycles to accomplish the same file transfer. This is a counter-intuitive result.

The newer code does complete more quickly, but by every measure it also severely monopolizes the CPU for long periods of time and uses more overall CPU time. The missing link that both of these things can be attributed to is the reduction in TCP timeout-based recoveries on the newer kernel because of the RFC 3517-related changes. The 2.6.22 code averaged 17 timeouts for each run of the test client. The 2.6.25 code averaged just 2. The result is graphed dramatically as much less idle ramp up time in between timeout events, resulting in less down time.

Fewer timeouts means the sender maintained a larger window on average. Large windows are necessary for decent throughput on high-latency links. This development version of the TCP stack had a significant advantage in transfer speed and finished 2 minutes quicker than the deployed stack largely because it was able to keep wider windows open.

However, those larger average windows also mean the SACK receiving code needs to do a lot more work for every packet received as the queue to be scanned contains more packets. The 2.7 billion packets scanned for the file transfer (4 times as many as the previous kernel version) and 10.07 sampled ticks per KB transferred are

testament to exactly how much work had to be done.

Faster processors do not help much in this situation, either. They would be able to scan larger packet chains in the same amount of time, but that in turn would simply expand the window a little bit, creating yet more work for each new option to be processed. A lot more processing cycles would be expended in order to process the same number of SACK options; the faster processor just creates more overhead for itself without getting any more real work done.

Conclusion

The performance implications of maliciously crafted SACK options can be very significant but do not rise to the level of a trivially exercisable DoS attack. The saving grace is the self-regulating nature of the occasional timeouts, but it is not hard to imagine a different client that could pace itself at a rate that occupied the server but did not overwhelm it to the point of timeout.

Computers that do not send large blocks of data do not need to be concerned about this, as they can never fill the large windows that are prerequisite for the exploit. While Selective Acknowledgments are critical for good performance on high bandwidth delay product network links, they are still an optional feature that can be disabled without sacrificing interoperability. The `sysctl` variable `net.ipv4.tcp_sack` can be set to 0 to disable SACK in the TCP stack.

Good work is going on for the general SACK handling case in the current Linux kernel development tree. This has laid the groundwork for future developments, such as packet list indices and partitions, which will help with some of the attack vectors down the road.

Resources

Learn

- Following are sources for quotes used in this article:
 - [David Miller](#)
 - [Ilpo Jarvinen \[1\]](#)
 - [Ilpo Jarvinen \[2\]](#)
 - [NC State University](#)
 - [CHC IT](#)
 - [Kode Vicious](#)
- [Simulation-based Comparisons of Tahoe, Reno, and SACK TCP \(PDF\)](#) by Kevin Fall and Sally Floyd is the seminal paper discussing the need for Selective Acknowledgments on high-delay paths.
- The main SACK patches referred to in this article are:
 - [Abstract tp->highest_sack accessing & point to next skb](#)
 - [Rewrite SACK block processing & sack_rcv_cache use](#)
 - [non-FAK SACK follows conservative SACK loss recovery](#)
- See also the original [10 patches](#), which form the bulk of the 2.6.25 SACK kernel changes.
- Read a discussion of [red-black trees](#) in the Linux Kernel.
- RFCs [2018](#), [2883](#), and [3517](#) are the Internet standards documents that govern SACK. They can be found in the IETF RFC archive.
- This IBM Redpaper provides [Linux Performance and Tuning Guidelines](#).
- In the [developerWorks Linux zone](#), find more resources for Linux developers, and scan our [most popular articles and tutorials](#).
- See all [Linux tips](#) and [Linux tutorials](#) on developerWorks.
- Stay current with [developerWorks technical events and Webcasts](#).

Get products and technologies

- [Order the SEK for Linux](#), a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- With [IBM trial software](#), available for download directly from developerWorks,

build your next development project on Linux.

Discuss

- Get involved in the [developerWorks community](#) through blogs, forums, podcasts, and community topics in our [new developerWorks spaces](#).

About the author

Patrick McManus

Patrick is an experienced [software engineer and entrepreneur](#) with a track record in multiple startup and emerging technology ventures. He specializes in close to the metal high bandwidth networked services.

Trademarks

DB2, Lotus, Rational, Tivoli, and WebSphere are trademarks of IBM Corporation in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.