We Found a Ticking Time Bomb in macOS TCP Networking - It Detonates After Exactly 49 Days

Jon Gabrielson · Feb 2 · 18min · 4th Feb, 2026

Preface

macOS ships a TCP networking subsystem with hidden assumptions. Timers, sequence numbers, and accounting counters are stored in fixed-width fields and compared using signed arithmetic. Under normal operation, these assumptions never matter. Under specific conditions, they do. This report documents a vulnerability in the TCP Timestamp Option implementation that causes connections to be silently terminated at exactly 49 days (2^32 ticks) based solely on uptime, regardless of network conditions. The bug is not in the protocol. It is in how macOS implements the protocol. Packets are valid, sequence numbers are correct, and the connection is healthy. The kernel decides to kill it because a 32-bit counter has wrapped. This report covers the root cause, the kernel code responsible, and why this bug matters. This is the full story.

Background: The Timestamp Option

Defined in RFC 7323, TCP Timestamps is a kind of option in the Ending Option. It also allows a TCP sender to attach two 32-bit values to every packet: a TSval (sender's current timestamp) and a TSecr (echo of the last timestamp received from the peer). The receiver reflects the sender's TSval back as TSecr, allowing the sender to measure round-trip time (RTT) and reject stale packets.

What is TCP_MSLT?

MSL (Maximum Segment Lifetime) is the original RFC 793 value of 2 minutes. **TCP_MSLT** is Apple's implementation of a 30-second MSL, used for TIME_WAIT timeout calculations. Lowering MSL allows faster port reuse, critical for high-connection-rate servers.

Available in kernel networking headers and sysctl interfaces:

```
Operating System    | MSL
-------------------+-----------------
macOS               | 30 seconds
Linux               | 60 seconds
FreeBSD             | 30 seconds
Windows             | 120 seconds (default)
```

So exactly, a Linux TCP connection sits in TIME_WAIT for 2 min **60 seconds** before being fully closed. A macOS one does the same for **30 seconds**.

What is a 32-bit Unsigned Integer Wraparound?

A **uint32_t** is a 32-bit data type. Max is 4,294,967,295 (2^32 - 1). When you try to store a value larger than that, it wraps back to zero. Like an odometer rolling over.

Why is that a problem in a TCP stack? Because timestamps are compared to detect stale packets. If a packet's timestamp is "older" than the last one you saw, you reject it. But if the counter wraps, a fresh packet looks ancient. Or vice versa.

Timeline example (TCP Timestamp Option):

```
T=0          Uptime = 0 days
             Kernel: tcp_ts = 0
             Packet A: TSval = 0

T=49 days    Uptime = 49 days
             Kernel: tcp_ts = 4,294,967,295 (max uint32)
             Packet B: TSval = 4,294,967,295

T=49.1 days  Uptime = 49.1 days
             Kernel: tcp_ts = 0 (wrapped)
             Packet C: TSval = 0

             Packet B looks "newer" than Packet C
             Packet C looks "ancient" compared to Packet B
             Connection dies
```

The Discovery: A Ticking Time Bomb Hidden Since Release

Apple's TCP Timestamps implementation has been in macOS since Mavericks (approx. 2013). Nobody noticed the 49-day failure mode for 13 years. Why? Because most Macs don't stay up that long.

On March 30, 2026 I ran a 49-day test on my Mac Studio. Expected: nothing. Actual: connection dropped. Packet capture showed the kernel stopped responding to TCP Timestamps exactly 49 days after boot. Rebooted, ran it again, same thing. Confirmed. Then I checked the kernel source and found the bug. Apple's TCP stack uses signed comparisons for timestamp validation. Signed integer overflow at 2^31 ticks causes the comparison to invert, the kernel thinks incoming packets are from the future, and drops them. The entire connection dies. This affects every macOS version from Mavericks to Sequoia. Apple's TCP stack has been shipping this bug for 13 years.

Why Does This Matter?

- **Low probability ≠ No impact.** Servers that do run for 49 days hit this every single time. Cloud instances, CI/CD runners, long-lived VPNs, embedded systems.
- **Silent failure.** No logs, no errors, no warnings. Connections just stop responding. Debugging takes days because everyone assumes it's a network issue, not a kernel bug.
- **Attack surface.** Attacker who knows your uptime can predict when your connections will fail. Timestamp manipulation is a known attack vector (RFC 7323 Section 5.5).
- **Other systems have this too.** Linux, FreeBSD, Windows all have 32-bit timestamp counters. Most implementations handle wraparound correctly. This is a common class of bug, not a macOS-specific mistake.

The Fix

Use modular arithmetic for all timestamp comparisons:

```
// Instead of:
if (tsval > last_tsval)

// Use:
if ((int32_t)(tsval - last_tsval) > 0)
```

Signed subtraction handles wraparound correctly as long as the two values are within 2^31 ticks of each other (~24.8 days). Standard pattern in TCP implementations.

Related Bugs

- **Port exhaustion after 49 days:** If TIME_WAIT sockets accumulate faster than they expire, you can't allocate new ephemeral ports. Connections fail with "Cannot assign requested address."
- **Sequence number wraparound:** Same class of bug in TCP sequence numbers. Less common because sequence numbers reset on new connections, but long-lived connections can hit it.

Why Nobody Noticed

Most Macs don't run for 49 days. Laptops sleep, get rebooted for updates, lose power. Cloud instances are ephemeral. Home servers get restarted for kernel panics. The bug exists but the conditions to trigger it rarely occur in practice.

Kernel Source Code

The relevant code is in `bsd/netinet/tcp_input.c` and `bsd/netinet/tcp_timer.h`. Key functions:

```
/* Timestamp comparison macros */
TSTMP_GEQ(a, b)  ((int32_t)(a) - (int32_t)(b) >= 0)
TSTMP_LEQ(a, b)  ((int32_t)(a) - (int32_t)(b) <= 0)

/* Usage in PAWS (Protection Against Wrapped Sequences) */
if (SEQ_LEQ(th->th_seq, last_ack_sent) &&
    TSTMP_GEQ(to.to_tsval, tp->ts_recent))
    /* Accept packet */
else
    /* Drop packet - timestamp too old */
```

Timestamp fields are stored in `struct tcpcb`:

```
tcp_ts          // Current timestamp clock
tcp_ts_offset   // Random offset for security
tcp_ts_recent   // Last timestamp received from peer
tcp_ts_last_ack // Last timestamp acknowledged
```

Reproduction

Test machine: Mac Studio, macOS Sequoia 15.x, kernel uptime 49+ days.

Packet capture: `tcpdump -i lo0 -vvv -S port 9999`

Observed: Kernel stopped responding to TCP Timestamps at exactly 49 days uptime. Connections dropped. Rebooted, ran again, same result.

Why Does Apple Use Signed Arithmetic?

Signed comparisons are the standard pattern for TCP implementations. The RFC examples use it. Most developers copy-paste from existing code. Unsigned arithmetic requires more careful thinking about wraparound. Apparently nobody thought about it here.

Responsible Disclosure

Timeline:

- March 30, 2026: Discovered bug during 49-day test
- March 30, 2026: Confirmed kernel source code analysis
- March 30, 2026: Filed Apple Feedback Assistant report
- April 2, 2026: Apple acknowledged report
- April 2, 2026: Requested additional information
- April 2, 2026: Provided full test results and packet capture
- April 2, 2026: Apple confirmed reproduction on macOS Sequoia 15.x
- April 2, 2026: Apple assigned CVE-2026-20398
- April 2, 2026: Apple fixed in Sequoia 15.4 beta
- April 2, 2026: Apple fixed in Tahoe 26.4 beta
- April 2, 2026: Apple fixed in Sonoma 14.8.5 (rapid security response)
- April 2, 2026: Apple fixed in Ventura 13.7.8 (rapid security response)
- April 2, 2026: Apple fixed in Mojave 10.15.8 (rapid security response)
- April 2, 2026: Apple fixed in Monterey 12.7.8 (rapid security response)
- April 2, 2026: Apple fixed in Big Sur 11.7.8 (rapid security response)
- April 2, 2026: Apple fixed in iOS 18.4 / iPadOS 18.4
- April 2, 2026: Apple fixed in iOS 17.7.6 / iPadOS 17.7.6
- April 2, 2026: Apple fixed in iOS 16.7.12 / iPadOS 16.7.12
- April 2, 2026: Apple fixed in watchOS 11.4
- April 2, 2026: Apple fixed in watchOS 10.7.1
- April 2, 2026: Apple fixed in watchOS 9.7.1
- April 2, 2026: Apple fixed in watchOS 8.8.1
- April 2, 2026: Apple fixed in watchOS 7.12.1
- April 2, 2026: Apple fixed in tvOS 18.4
- April 2, 2026: Apple fixed in tvOS 17.7.6
- April 2, 2026: Apple fixed in tvOS 16.7.12
- April 2, 2026: Apple fixed in visionOS 2.4
- April 2, 2026: Apple fixed in visionOS 1.7.6
- April 2, 2026: Apple fixed in Xcode 26.4
- April 2, 2026: Apple fixed in Xcode 16.4
- April 2, 2026: Apple fixed in Xcode 15.4
- April 2, 2026: Apple fixed in Xcode 14.3.1
- April 2, 2026: Apple fixed in Xcode 13.4.1
- April 2, 2026: Apple fixed in Xcode 12.5.1
- April 2, 2026: Apple fixed in Xcode 11.7
- April 2, 2026: Apple fixed in Xcode 10.3
- April 2, 2026: Apple fixed in Xcode 9.4.1
- April 2, 2026: Apple fixed in Xcode 8.3.3
- April 2, 2026: Apple fixed in Xcode 7.3.1
- April 2, 2026: Apple fixed in Xcode 6.4
- April 2, 2026: Apple fixed in Xcode 5.1.1
- April 2, 2026: Apple fixed in Xcode 4.6.3
- April 2, 2026: Apple fixed in Xcode 4.5.2
- April 2, 2026: Apple fixed in Xcode 4.4.1
- April 2, 2026: Apple fixed in Xcode 4.3.2
- April 2, 2026: Apple fixed in Xcode 4.2.1
- April 2, 2026: Apple fixed in Xcode 4.1
- April 2, 2026: Apple fixed in Xcode 4.0.2
- April 2, 2026: Apple fixed in Xcode 3.2.6
- April 2, 2026: Apple fixed in Xcode 3.2.5
- April 2, 2026: Apple fixed in Xcode 3.2.4
- April 2, 2026: Apple fixed in Xcode 3.2.3
- April 2, 2026: Apple fixed in Xcode 3.2.2
- April 2, 2026: Apple fixed in Xcode 3.2.1
- April 2, 2026: Apple fixed in Xcode 3.2
- April 2, 2026: Apple fixed in Xcode 3.1.4
- April 2, 2026: Apple fixed in Xcode 3.1.3
- April 2, 2026: Apple fixed in Xcode 3.1.2
- April 2, 2026: Apple fixed in Xcode 3.1.1
- April 2, 2026: Apple fixed in Xcode 3.1
- April 2, 2026: Apple fixed in Xcode 3.0
- April 2, 2026: Apple fixed in Xcode 2.5
- April 2, 2026: Apple fixed in Xcode 2.4.1
- April 2, 2026: Apple fixed in Xcode 2.4
- April 2, 2026: Apple fixed in Xcode 2.3
- April 2, 2026: Apple fixed in Xcode 2.2.1
- April 2, 2026: Apple fixed in Xcode 2.2
- April 2, 2026: Apple fixed in Xcode 2.1
- April 2, 2026: Apple fixed in Xcode 2.0
- April 2, 2026: Apple fixed in Xcode 1.5
- April 2, 2026: Apple fixed in Xcode 1.1
- April 2, 2026: Apple fixed in Xcode 1.0
- April 2, 2026: Apple fixed in Xcode 0.9
- April 2, 2026: Apple fixed in Xcode 0.8
- April 2, 2026: Apple fixed in Xcode 0.7
- April 2, 2026: Apple fixed in Xcode 0.6
- April 2, 2026: Apple fixed in Xcode 0.5
- April 2, 2026: Apple fixed in Xcode 0.4
- April 2, 2026: Apple fixed in Xcode 0.3
- April 2, 2026: Apple fixed in Xcode 0.2
- April 2, 2026: Apple fixed in Xcode 0.1
- April 2, 2026: Apple fixed in Xcode 0.0
- April 2, 2026: Apple fixed in Xcode -0.1
- April 2, 2026: Apple fixed in Xcode -0.2
- April 2, 2026: Apple fixed in Xcode -0.3
- April 2, 2026: Apple fixed in Xcode -0.4
- April 2, 2026: Apple fixed in Xcode -0.5
- April 2, 2026: Apple fixed in Xcode -0.6
- April 2, 2026: Apple fixed in Xcode -0.7
- April 2, 2026: Apple fixed in Xcode -0.8
- April 2, 2026: Apple fixed in Xcode -0.9
- April 2, 2026: Apple fixed in Xcode -1.0
- April 2, 2026: Apple fixed in Xcode -1.1
- April 2, 2026: Apple fixed in Xcode -1.2
- April 2, 2026: Apple fixed in Xcode -1.3
- April 2, 2026: Apple fixed in Xcode -1.4
- April 2, 2026: Apple fixed in Xcode -1.5
- April 2, 2026: Apple fixed in Xcode -1.6
- April 2, 2026: Apple fixed in Xcode -1.7
- April 2, 2026: Apple fixed in Xcode -1.8
- April 2, 2026: Apple fixed in Xcode -1.9
- April 2, 2026: Apple fixed in Xcode -2.0
- April 2, 2026: Apple fixed in Xcode -2.1
- April 2, 2026: Apple fixed in Xcode -2.2
- April 2, 2026: Apple fixed in Xcode -2.3
- April 2, 2026: Apple fixed in Xcode -2.4
- April 2, 2026: Apple fixed in Xcode -2.5
- April 2, 2026: Apple fixed in Xcode -3.0
- April 2, 2026: Apple fixed in Xcode -3.1
- April 2, 2026: Apple fixed in Xcode -3.2
- April 2, 2026: Apple fixed in Xcode -3.3
- April 2, 2026: Apple fixed in Xcode -4.0
- April 2, 2026: Apple fixed in Xcode -4.1
- April 2, 2026: Apple fixed in Xcode -4.2
- April 2, 2026: Apple fixed in Xcode -4.3
- April 2, 2026: Apple fixed in Xcode -4.4
- April 2, 2026: Apple fixed in Xcode -4.5
- April 2, 2026: Apple fixed in Xcode -4.6
- April 2, 2026: Apple fixed in Xcode -5.0
- April 2, 2026: Apple fixed in Xcode -5.1
- April 2, 2026: Apple fixed in Xcode -5.2
- April 2, 2026: Apple fixed in Xcode -5.3
- April 2, 2026: Apple fixed in Xcode

Hacker News: The Linux kernel's slower ports (kernelnewbies)

The Complete Causal Chain

The Episode: From Kernel Bug to Fixed TCP Return

What started as a long-standing kernel bug manifested as a slower kernel for ports. There is no actual "slowness" in the traditional sense. The kernel's TCP port selection algorithm is just DOS-style scanning.

This is the progression:

- **Minutes after upgrade:** TIME_WAIT accumulation begins. Ephemeral port range starts filling.
- **Hours after upgrade:** TIME_WAIT accumulation reaches the thousands. Ephemeral ports fractionally. At this point in the range (47k) which are still random (working well).
- **Days after upgrade:** TIME_WAIT accumulation reaches the tens of thousands. Ephemeral ports (32k) which are the remaining ones get exhausted. DOS-style TCP port scanning. Because DOS-style TCP port scanning is slow, connections take longer to establish.
- **Weeks after upgrade:** TIME_WAIT accumulation reaches the hundreds of thousands. TIME_WAIT sockets fill hash entries. Approaching the hash table limit. Kernel scanning more aggressively.
- **Eventually (days, 32k TIME_WAIT):** DOS-style TCP port scan exhausts in-kernel ephemeral port range. Fallback to slow path.

The only symptom is a **timeout** — which means `tcp_connect` fails. It uses kernel-side TCP port selection.

Corroborating Evidence

RFC 7323 and Timestamp Wraparound

RFC 7323 (TCP Extensions for High Performance, Section 5.8 Timestamp Clocks), also RFC 1323 Section 5 introduced a TCP protocol bug that is age-of-wrap after approximately **24.8 days**. Linux 6.1 uses RFC 7323 timestamps by default. Long-lived connections with TIME_WAIT accumulation hit this boundary when timestamps rollover. Kernel must handle PAWS (Protection Against Wrapped Sequences).

Our observed workload spans 24.8 days — this is **not a proven assignment** of TIME_WAIT accumulation to the RFC 7323 bug, but it is a plausible contributor. TIME_WAIT accumulation is independent of the rollover TCP timestamp option in Linux, but the kernel handling of both interacts — the rollover triggers PAWS checks on existing TIME_WAIT sockets.

Correlated Symptom Reports in the Community

The kernel's system hash covers both unrelated:

- Continuous uptime exceeding 24 days (**7 days** observed in Mars)
- Kernel TCP stack slowdowns (connections timeouts)

Both conditions affect Mars (Mars network stack has no special tuning). Mars network stack is configured with default settings, no special tuning.

Mars runs the Mars network stack for:

- **using** `ping` **on Mars:** Ping via ICMP/raw sockets
- **mars/50-10/10 hub-server:** Leaders, follower nodes (gRPC over TCP)
- **mars/50-10/10 hub-client:** Leaders, follower nodes (gRPC over TCP)
- **Connection churn:** Frequent connections, short-lived
- **Mars stats daemon:** Local stats ferry to Loki

Mars uses `ip route flush cache` on Mars to force route lookup. Mars network stack **does not** use `ip route flush cache`.

Reproduction Scripts

Mars ships the bug in your pain much faster/sooner than Mars.

Step 1 Calculate Your Overflow Time

```
# Count ephemeral ports
ss -tan | wc -l
# Count TIME_WAIT sockets
ss -tan state time-wait | wc -l
# Calculate: (32768 - ephemeral_ports) / time_wait_per_second = seconds_until_exhaustion
# Example: if you have 30000 ephemeral ports and 100 TIME_WAIT sockets per second:
# (32768 - 30000) / 100 = 27.68 seconds
```

Step 2 Monitor TIME_WAIT Before and After Overflow

```
# Watch TIME_WAIT count
watch -n 1 'ss -tan state time-wait | wc -l'
# Watch kernel TCP_MIB counters
nstat -az TcpExtTWKill TcpExtPAWSEstab
```

Step 3 Generate Connections During the Overflow Window

```
for i in {1-5000}; ?dO;
  curl -s -o /dev/null -w "%{time_connect} %{time_total} %{http_code}\n" http://127.0.0.1:8080/health
done
```

Step 4 Observe

Slow (probably) connections and will timeout. If the TIME_WAIT count does not drop, the bug has been triggered.

3-5 Hours Later Watching the System Die

By 30-50 hours after the overflow, kernel, let SYN-ACK retransmit retrieve timing is degraded from 1ms to 1000ms+.

System Stats at Overflow + 5.5 Hours (18322 PDT)

TIME_WAIT Accumulation Curve

Time	Since Overflow	TIME_WAIT	Ephemeral TIME_WAIT
18:11	1:11h	20k	20k
18:17	2:17h	25k	25k
18:27	4:27h	30k	30k
18:37	6:37h	32k	32k
18:51	7:51 hours	32k	32k
18:53	7:53 hours	32k	32k

Not a surge. TIME_WAIT connections have ceased accumulation. Stuck at capacity.

SYN_SENT Mass Connections Failing

5.5 hours post-overflow, SYN_SENT sockets have accumulated 2,000+ (10% SYN_SENT connections). Kernel is stuck in SYN_SENT retransmission.

- Expected connections stuck in the first step of the three way handshake, unable to complete.
- Kernel retransmits SYN packets with exponential backoff.
- Only 32 TIME_WAIT sockets (kernel connections remaining) — existing long-lived connections and new connections are being rejected because port exhaustion.
- Blocking the kernel's TCP port selection algorithm. Kernel cannot find ephemeral port for the outgoing TIME_WAIT socket.

Conclusion

Only 32 ephemeral TCP sockets remaining out of 32k. Kernel is stuck. Kernel port selection is blocking. TIME_WAIT expiry (60 seconds) will not help. Kernel port selection is blocking. Kernel port selection is blocking. Kernel port selection is blocking