Seven TLS 1.3 libraries never rotate session keys

one key unlocks every record — harvest now, decrypt later

`Discovered 8 Apr 2026 · Edited 19 May 2026 · Updated 11 Jul 2026`

> Learned after the finding: the class splits maintainers. wolfSSL took Cybernuke's fix and shipped it in 5.9.2 (23 Jun 2026); Oracle had closed the same gap in Java years earlier as CVE-2023-21930. OpenSSL's CTO and a Go security-team maintainer decline to treat the absence as a vulnerability, holding that key rotation is the application's job. Cybernuke's answer is that a transport-layer protocol exists so the application above need not manage the channel's keys, and that the standard has since hardened: RFC 9846 (11 July 2026, obsoleting RFC 8446) raises SHOULD rekey to MUST rekey (or close the connection).

A backup runs overnight. A database replica streams across a region. A trading session holds its socket from the opening bell. Ordinary, sanctioned, encrypted. TLS 1.3 was built to retire each key as the bytes pile up and derive a fresh one, so a stolen key opens what follows it and nothing before. Seven libraries never do it. The connection runs for a day, a month, a year on one unchanging secret, and when that secret falls, the attacker doesn't just read the history. The key that decrypts is the key that forges; they can inject into the live stream and take the session over. Not what follows the theft. All of it, back to the handshake.

> *Traffic key derived once, never replaced — seven libraries run weak cryptography.*

TLS 1.3 seals every record of a connection under a traffic key derived once, at the handshake. That key has a budget. RFC 8446 §5.5 puts it at 2^24.5 full-size records, some 389 GB, beyond which AES-GCM's guarantees start to erode. The protocol's remedy is KeyUpdate: a post-handshake message that swaps in a fresh key and retires the old one. It does two jobs at once. It keeps the AEAD cipher inside its proven safety limit, and it bounds forward secrecy: with the previous key destroyed, a secret stolen later decrypts only records sealed since the last rotation, not the whole session.

A connection that never rotates forfeits both at once. One stolen traffic secret (32 bytes lifted from a core dump, a swapped page, a Heartbleed-class read) decrypts every application record the connection ever carried, past and future. On a long-lived link that is the entire session. And because AES-GCM is symmetric, the key that decrypts also forges: from an on-path position the holder injects into the live stream and takes the session over, not merely reads it. Harvest now, decrypt later.

Seven production TLS 1.3 libraries never send KeyUpdate on their own initiative: BoringSSL, Go's `crypto/tls`, LibreSSL, mbedTLS, OpenSSL, Tongsuo, and wolfSSL. The libraries that do rotate (GnuTLS and picotls were both observed doing so, in prior record) keep the traffic sent before a theft out of reach; the seven leave all of it in reach. They fail in four shapes: wolfSSL ships the trigger but mis-encodes the limit that arms it, so it never fires; OpenSSL, BoringSSL, and Tongsuo ship no automatic trigger but expose a manual KeyUpdate the application never calls; Go and LibreSSL expose no way to originate one at all, only to answer a peer's; and mbedTLS ships no KeyUpdate whatsoever: the message is unimplemented.

Don't trust, verify: Record limit passed, key unchanged

wolfSSL's own example client streams to stock OpenSSL, past 23,726,566 records, the count wolfSSL's source names as its AES-GCM limit.

```
# Terminal 1 -- stock OpenSSL, logging record headers and KeyUpdate.
$ sleep infinity | openssl s_server -msg -rev -accept 11111 -tls1_3 \
    -cert wolfssl/certs/server-cert.pem -key wolfssl/certs/server-key.pem \
    -keylogfile cross.keys 2>&1 \
  | grep --line-buffered -E '^<<<.*RecordHeader|KeyUpdate' > cross.log &

# Terminal 2 -- the client under test: wolfSSL's own echoclient, one
# record per line, asked nothing beyond the stream itself.
$ yes A | head -n 24000000 \
  | ( cd wolfssl && ./examples/echoclient/echoclient /dev/stdin /dev/null )

# Terminal 1, once the stream ends. Records it logged, against the count
# wolfSSL's own trigger names:
$ awk -v L=23726566 '/RecordHeader/{n++} \
      END{print(n>=L?"past the ceiling":"short")}' cross.log
past the ceiling

# KeyUpdate messages it saw, in either direction:
$ grep -c KeyUpdate cross.log
0

# and the application traffic secrets the connection ever had:
$ awk '/TRAFFIC_SECRET/ && !/HANDSHAKE/ {print $1}' cross.keys | sort
CLIENT_TRAFFIC_SECRET_0
SERVER_TRAFFIC_SECRET_0
```

Those are the two secrets derived at the handshake, one per direction, and the connection never created another: no KeyUpdate crossed either way, so every record it carried was sealed under that one pair.

```
# the crossing above ran on echoclient, which takes a file and no flags: of
# wolfSSL's two example clients it is the one that can be fed 24 million
# records, and it has no way to ask for a rotation. So the same experiment
# runs again here on the other client, which does take flags, twice.

# asked nothing, as above:
$ timeout 30 ./examples/client/client -h 127.0.0.1 -p 11111 -v 4
#   KeyUpdate the OpenSSL server logged, from the same -msg output the
#   crossing above is read from:  0
#   traffic secrets logged:
$ awk '/TRAFFIC_SECRET/ && !/HANDSHAKE/ {print $1}' plain.keys | sort
CLIENT_TRAFFIC_SECRET_0
SERVER_TRAFFIC_SECRET_0

# and the same command, one stock flag apart. The flag is -I, the library's
# own "update keys and IVs before sending data":
$ timeout 30 ./examples/client/client -h 127.0.0.1 -p 11111 -v 4 -I
#   KeyUpdate the OpenSSL server logged:  2
#   traffic secrets logged -- _N is OpenSSL's own label for one that a
#   rotation installed, written at ssl/tls13_enc.c:824:
$ awk '/TRAFFIC_SECRET/ && !/HANDSHAKE/ {print $1}' rotate.keys | sort
CLIENT_TRAFFIC_SECRET_0
CLIENT_TRAFFIC_SECRET_N
SERVER_TRAFFIC_SECRET_0
SERVER_TRAFFIC_SECRET_N

# so the absence above is wolfSSL's own policy, not a blind detector and not
# an unimplemented message: the same library rotates the moment it is asked.
```

The above shows wolfSSL can rotate, when instructed, and that nothing instructed it in the session that ran past the limit. The other six never ship the trigger at all: a sweep drove ten implementations past that same limit in both roles, with GnuTLS and picotls rotating as working positive controls.

Trace to vulnerability site: Limit missed by ×65k

The session above ran past 23,726,566 records with wolfSSL's rotation code compiled in and never reached. A switch on the negotiated cipher picks the AES-GCM usage limit, the logic is correct, and everything rides on `AEAD_AES_LIMIT`, read at the `v5.8.4-stable` tag, commit `59f4fa5`:

```
# wolfSSL is the only one of the seven that implements the trigger at all,
# and this is the branch a TLS_AES_256_GCM_SHA384 session takes:
$ sed -n 25599,25601p src/internal.c
        case wolfssl_aes_gcm:
            /* Limit is 2^24.5 */
            limit = AEAD_AES_LIMIT;

# the limit, at its line in the pinned tree:
$ sed -n 1423,1426p wolfssl/internal.h
/* Limit is 2^24.5
 * https://www.rfc-editor.org/rfc/rfc8446#section-5.5
 * Without the fraction is 23726566 (0x016A09E6) */
#define AEAD_AES_LIMIT                           w64From32(0x016A, 0x09E6)

# and w64From32's own body, which is what makes the split wrong:
$ sed -n 1029p wolfcrypt/src/misc.c
    ret.n = ((word64)hi << 32) | lo;

# so the trigger is armed at:      1554778163686 records
# where the comment above it says: 23726566 records

# the same mis-split, in the DTLS limits below it (alignment collapsed to fit):
  1443:#define DTLS_AEAD_AES_CCM_FAIL_LIMIT w64From32(0x00B5, 0x04F3)
  1444:#define DTLS_AEAD_AES_CCM_FAIL_KU_LIMIT w64From32(0x005A, 0x8279)
```

The literal is right: `0x016A09E6` is exactly 23,726,566, the integer part of 2^24.5. The encoding is wrong. `w64From32(hi, lo)` builds `(hi << 32) | lo`, so splitting the 32-bit literal into 16-bit halves `0x016A` and `0x09E6` shifts the high half by 32 bits, not 16, yielding `0x016A_0000_09E6` ≈ 2^40.5. The trigger is armed at a limit about 65,000× too high.

The other six carry the remaining three shapes.

- OpenSSL, BoringSSL, and Tongsuo (an OpenSSL derivative) expose `SSL_key_update()` as public API and never wire it to a counter or a threshold. BoringSSL is the sharpest case: it does the record-layer work and leaves rotation policy to whoever embeds it, and neither fills the gap. The only callers in its tree are its own tests.
- Go and LibreSSL cannot originate one at all. Go's `crypto/tls` exposes no rotation API whatsoever: `Conn` carries no KeyUpdate method (Kubernetes, etcd, and gRPC cannot call what does not exist), and the only recourse is to tear the connection down and handshake afresh, the antithesis of a long-lived stream. LibreSSL can only answer: `tls13_key_update_recv` receives a peer's KeyUpdate, and the sole send it has is the reply that peer compels from it.
- mbedTLS 3.6.5 carries no KeyUpdate at all: the source has no post-handshake handler for it, so mbedTLS can neither originate a rotation nor answer a peer's. Where mbedTLS ships in firmware, there is no in-connection forward secrecy to be had at any layer.

TLS abbreviates Transport Layer *Security*: the point of a transport-layer protocol is that the application above never reaches down to manage the channel's keys. Yet where these libraries let a key rotate at all, they leave the call to the application (which no transport-layer application makes) and Go, LibreSSL, and mbedTLS do not offer even that.

Fix

wolfSSL's is a one-line correction: land the record-count literal wholly in the `w64From32` low word, so the trigger fires at 2^24.5 instead of 2^40.5.

```
--- a/wolfssl/internal.h
+++ b/wolfssl/internal.h
@@ -1420,10 +1420,11 @@ enum {
 #endif


-/* Limit is 2^24.5
- * https://www.rfc-editor.org/rfc/rfc8446#section-5.5
- * Without the fraction is 23726566 (0x016A09E6) */
-#define AEAD_AES_LIMIT                           w64From32(0x016A, 0x09E6)
+/* AES-GCM confidentiality limit, RFC 8446 section 5.5: 2^24.5 records.
+ * 2^24.5 has a fractional exponent, so it cannot be a (1 << n) shift --
+ * it is the literal 23726566 (0x016A09E6), which must occupy the low
+ * word of w64From32(hi, lo) in full; hi is 0. */
+#define AEAD_AES_LIMIT                           w64From32(0, 0x016A09E6)
 /* Limit is 2^23
  * https://www.rfc-editor.org/rfc/rfc9147.html#name-integrity-limits */
 #define DTLS_AEAD_AES_CCM_LIMIT                  w64From32(0, 1 << 22)
```

Applied to the pinned checkout and rebuilt:

```
# apply the guard to the pinned checkout and rebuild it
$ git apply -p1 fix.patch
$ ./autogen.sh
$ ./configure --enable-tls13
$ make -j$(nproc)
```

Then the session at the top of this page runs a second time, unchanged, against the rebuilt library:

```
# Terminal 1 is the same server line as above, with cross.keys and
# cross.log renamed fixed.*; nothing else in the rig is touched.

# Terminal 2 -- the same client, the same stream, one patch apart:
$ yes A | head -n 24000000 \
  | ( cd wolfssl && ./examples/echoclient/echoclient /dev/stdin /dev/null )

# records the server logged, against the same count:
$ awk -v L=23726566 '/RecordHeader/{n++} \
      END{print(n>=L?"past the ceiling":"short")}' fixed.log
past the ceiling

# and this time the library rotates, unprompted -- counting the records
# logged up to the first KeyUpdate gives the index it fired at:
$ awk -v L=23726566 '/RecordHeader/{r++} \
      /KeyUpdate/{ if (r>=L) print "rotated at the limit" \
      ; else print "rotated early"; f=1; exit } \
      END{ if (!f) print "never rotated" }' fixed.log
rotated at the limit

# the application traffic secrets the connection had:
$ awk '/TRAFFIC_SECRET/ && !/HANDSHAKE/ {print $1}' fixed.keys | sort
CLIENT_TRAFFIC_SECRET_0
CLIENT_TRAFFIC_SECRET_N
SERVER_TRAFFIC_SECRET_0
SERVER_TRAFFIC_SECRET_N
```

Same command, same span, one constant apart (`w64From32(0, 0x016A09E6)` in place of `w64From32(0x016A, 0x09E6)`).

The honest cost. wolfSSL had the scaffold (a record counter, a limit check, a call into KeyUpdate) and only the constant was wrong; the other six ship no scaffold, so their fix is larger. It means a new field in record-layer state, an increment per sealed record, a threshold check, and an internal call into the KeyUpdate sender, roughly 30–80 lines, landing in a different place per library. LibreSSL needs the least: its `ssl/tls13_record_layer.c` already increments the AEAD sequence number per record, so only the threshold check and the self-initiated send are missing. mbedTLS needs the KeyUpdate message built first.

Scope

The precondition is a single, long-lived TLS 1.3 connection that pushes sustained volume without being torn down. RFC 8446 §5.5 bounds *plaintext* and states the bound as 2^24.5 full-size records, which is where 389 GB comes from; a library implementing that as a record counter, as wolfSSL does, reaches its own trigger sooner on a stream of small messages, whatever they carry.

QUIC and HTTP/3 are out entirely: RFC 9001 key update is a separate mechanism that implementations *do* fire. What remains is exactly the world of pinned, continuously-streamed connections: replication links, telemetry pipelines, and long-lived feeds.

Crossing 389 GB forfeits *forward secrecy*, and it happens in hours. That loss arrives in every scenario below. In markets, what those sessions carry is not public quote data but the subscriber's own activity: order flow, positions, the instruments they watch, all of it under the one key.

How long these connections take to pass the standard's own limit is the second half. It is measured at full-size records, the slowest any connection reaches it, against sustained rates a single connection carries:

single persistent connection	sustained rate	crosses 389 GB
market-data WebSocket feed	~2 MB/s	~54 h
proprietary feed	~5 MB/s	~22 h
DB replication	~100 MB/s	~65 min
log / telemetry firehose	~250 MB/s	~26 min
storage replication	1.25 GB/s	~5 min
datacentre fabric	12.5 GB/s	~31 s

The honest bound: this is not "plaintext recovered in a day." At the ~2^64-block birthday event — 2^68 B ≈ 295 EB, centuries at any real bandwidth — the proof runs out: past it nothing rules out an attacker telling the ciphertext from random, which is a long way short of reading it. The claim is narrower and firmer: one key seals the connection from its first record, so a secret stolen at any moment opens every record that connection ever carried. Volume adds only the second half, the *provable* margin eroding to inadequate in days at datacentre rates, on a key the library was never told to rotate. RFC 8446 §5.5 says implementations SHOULD rekey before the limit; RFC 9846 raises that to a MUST, satisfied by rekeying *or* by closing. Seven libraries reach the limit and do neither.

Afterword: a MUST that lives in one document

RFC 8446 §5.5 says implementations SHOULD rekey before 2^24.5 records; RFC 9846 raises it to a MUST. Seven libraries don't, and what follows is the cost — the honest answer is not the one the limit suggests.

Past the limit the guarantee decays quadratically, which means leakage, not collapse. The authors of the analysis the standard cites said so when the working group asked: *"confidentiality will not collapse, the limit indicates the moment when information starts leaking, which could start off as little as one bit of plaintext."* And against the opposite error: *"beyond the bounds, there are no guarantees."* Recovery is real, needs 9.4 ZB under one key (Leurent and Sibleyras, 2018). So the limit is set by the proof, not the attack.

Then the question is who the MUST is for. NIST's own limit for this cipher is half a billion times higher. Germany's TLS guideline does not contain the string KeyUpdate; NIST SP 800-52r2 runs 3,865 lines and names no key lifetime; the NSA's advisory neither. Validation cannot reach it and says so, the Galois/Counter Mode Validation System testing *"conformance to GCM, GMAC and XPN specifications rather than provide a measure of a product's security"*. Nor has anyone machine-checked the number.

So a maintainer has two choices, both defensible. The margin is half a billion times more conservative than NIST's, it was made mandatory, and nobody outside the working group noticed; on that reading, ignoring it costs nothing. Or: it says MUST, and a library that claims to implement TLS 1.3 does.

Neither choice touches why this finding exists. One traffic secret seals the whole connection, and rotation is what stops a key stolen at any moment from opening everything before it. Going right back to the first record.

IETF mandates forward secrecy; seven libraries don't (yet) deliver.

Discovered 2026-04-08; disclosed 2026-05-01, coordinated through CERT/CC (multi-library class as VRF#26-05-TRXPT, mbedTLS as VRF#26-03-GYXLX). Targets: BoringSSL on main at discovery, Go `crypto/tls` 1.23, LibreSSL 4.2.1, mbedTLS 3.6.5, OpenSSL 3.5.4, Tongsuo 8.4.0, wolfSSL 5.8.4. CWE-325 · CVSS 7.4 High (AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:N).

Detected, exploited, & patched by cybernuke — cybernuke.bensmyth.com.

`Edited 19 May 2026 — BoringSSL joins the affected set, having been incorrectly listed in the disclosure as a positive control: noticed by Go, confirmed on a rebuilt wire harness at 25 million records in both roles, and added to the case by CERT/CC.`

`Appendix: standing it up`

`Everything a maintainer needs is above. What follows is the rig. Clone the release tag rather than a tree named after a version, since master carries the last released version string until the next bump:`

```
$ git clone --depth 1 --branch v5.8.4-stable \
     https://github.com/wolfSSL/wolfssl.git wolfssl

# the tag, and the commit it resolves to:
$ git describe --tags
v5.8.4-stable
$ git rev-parse --short=7 HEAD
59f4fa5
```

`Both clients under test come out of that checkout, and the observer beside them is whatever OpenSSL the reader already has:`

```
# build the clients under test from the tag cloned above:
$ ( cd wolfssl && ./autogen.sh && ./configure --enable-tls13 \
      && make -j$(nproc) )

# these two binaries are the SUBJECT, and the version they report is
# asserted against the pinned tag:
$ ./examples/client/client --help | head -1 | cut -d' ' -f1-3
wolfSSL client 5.8.4
$ git describe --tags && ls examples/client/client \
      examples/echoclient/echoclient
v5.8.4-stable
examples/client/client
examples/echoclient/echoclient

# the capture needs a raw socket, so the rig runs in a rootless namespace:
# unshare -Urn maps you to root inside it and grants nothing on the host
$ unshare -Urn bash
$ ip link set lo up

# stock OpenSSL is the INSTRUMENT, never the subject: it terminates the
# connection so it can log the traffic secrets BOTH ends derive, which is
# how a KeyUpdate is detected without reading the library's own logs
$ openssl version
OpenSSL 3.5.5 27 Jan 2026 (Library: OpenSSL 3.5.5 27 Jan 2026)
```