

TLS 1.3, session resumption, 0-RTT early data, external PSKs and all that...

a journey down a rabbit hole

Viktor Dukhovni
OpenSSL Conference 2026 · Prague

Roadmap

1. My motivation — anonymous TLS in SMTP
2. TLS 1.3: PSK as the unifying abstraction
3. PSK brittleness — and RFC 9258
4. External PSK benefits and requirements
5. OpenSSL PSK implementation features and issues
6. Next steps
7. Questions?

My Motivation

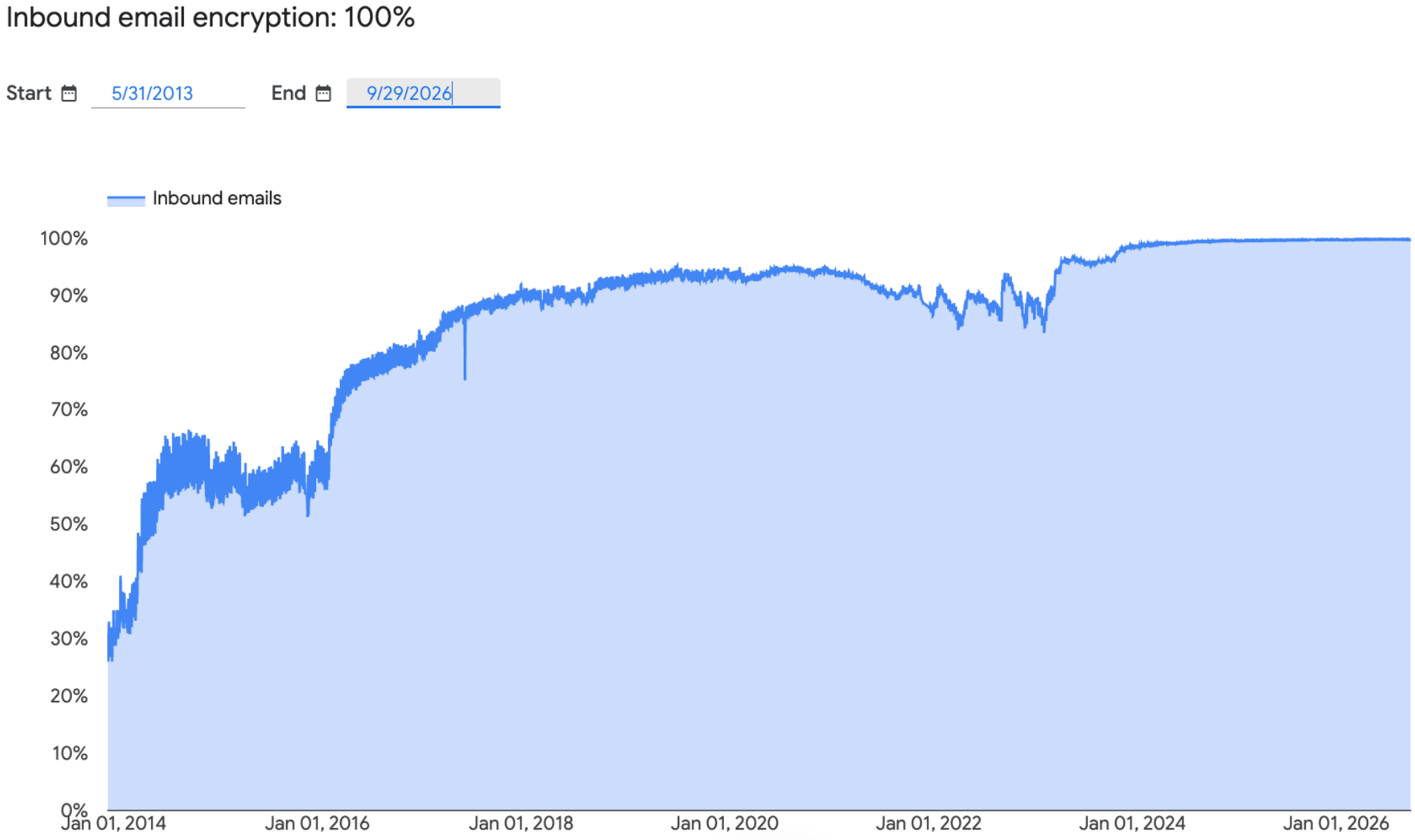
Anonymous TLS in SMTP

TLS in (MTA-to-MTA) SMTP

- **Opportunistic and effectively unauthenticated —** certificates are routinely self-signed, expired, or carry unrecognised names; any failure falls back to cleartext
- **Every handshake goes through the motions of authentication nobody checks —** a certificate chain transmitted, a signature made and verified: required by the protocol, ignored by the peer
- **TLS 1.2 can skip certificate authentication (**but TLS 1.3 has no DH_anon ciphers**)**:

```
Anonymous TLS connection established from mailer2.gandi.net[217.70.182.74]:
TLSv1.2 with cipher ADH-AES256-GCM-SHA384 (256/256 bits)
```

- **Yet TLS use is nearly universal —** inbound to Gmail 100%, outbound 97–99%



Google Email encryption in transit 2026/09/30

Anonymous TLS 1.3 — no new mechanisms required

PSK — a "pre shared key provisioned out of band

In TLS PSKs (**RFC42979**) are an alternative to certificate authentication

Available since TLS 1.0

If the "shared secret" is actually fixed and "well known"

Used together with asymmetric (Diffie-Hellman, ECDH, ...) key exchange

The result is analogous to TLS 1.2 with an ADH cipher

No new TLS 1.3 protocol machinery should be required

emailcore list, 2 July 2026

A few hours' work, surely?

- OpenSSL has had external-PSK callbacks since 1.1.1
- Wire a fixed, well-known key into Postfix; force `psk_dhe_ke`; disable early data

No new OpenSSL API.

Tried it, but...

SHA-384 handshake. SHA-256 PSK. Silently ignored.

— to explain that, we need to back up: what is a TLS 1.3 PSK?

openSSL CORPORATION

TLS 1.3

PSK as the unifying abstraction

PSKs took on multiple roles

- **External PSKs** — a key provisioned out of band
- **Resumption** — session IDs and session tickets, now just another PSK identity

  Early data (0-RTT) — the client sends a request in its first flight, before the server has responded

  The payload should be an idempotent request (preventing replays is difficult)

All under a single generalised PSK mechanism.

0-RTT data is encrypted under a key derived from the PSK — recognise the identity, decrypt the data.

TLS 1.3: more new machinery

- **Key exchange modes** — `psk_ke` (PSK alone), or `psk_dhe_ke`, which adds forward secrecy
  - "dhe" is historical — the mode now covers KEMs too (§4.3.9)
- **Binders** — the client MACs the ClientHello with a key derived from the PSK
  - the MAC digest is that of the negotiated ciphersuite — a PSK is no longer portable across ciphers
  - the server picks one identity and validates only that binder; if it fails the handshake aborts (§4.3.11)
- **HelloRetryRequest** — 2nd ClientHello (CH2) may drop PSKs and recompute binders, but not add any
  - and early data is not permitted after an HRR at all (§4.2.2)

TLS 1.3: PSK API (session-bearing callback)

```
/* client */
typedef int (*SSL_psk_use_session_cb_func)(SSL *ssl,
        const EVP_MD *md,   /* NULL except after HRR */
        const unsigned char **id, size_t *idlen,
        SSL_SESSION **sess);

/* server */
typedef int (*SSL_psk_find_session_cb_func)(SSL *ssl,
        const unsigned char *identity, size_t identity_len,
        SSL_SESSION **sess);
```

Return a fully built SSL_SESSION — cipher, version and master key

The session carries a cipher — and the cipher fixes the digest.

0-RTT is keyed by a (typically resumption) PSK

0-RTT needs a secret that predates the connection, so PSKs were extended to support it — tickets and external PSKs alike. Five conditions follow:

- Early data is keyed by the first PSK identity offered (RFC 9846 §4.3.10)
- The TLS version must match the one recorded on the PSK
- The server must negotiate exactly the PSK's ciphersuite to accept it
  - resumption itself needs only the same KDF digest
- The ALPN protocol must match the one recorded on the session

Anti-replay is the application's problem (§8)

No anti-replay with an external PSK

All decided client-side, before server response.

Same code path, different rules

	1-RTT (any PSK)	0-RTT early data
Hash	must match	must match
Ciphersuite	any with same hash	**exact match**
ALPN	may change	must match
Which identity	any offered	**first offered only**

Every bold cell on the right is a condition the client must get right before it sends a byte.

PSK brittleness

and RFC 9258

> “The protocol mandates that each PSK only be used with a single hash function. **This was done to simplify protocol analysis.**”
>
> RFC 9258 §1

This hampers deployment and complicates implementation.
“The formal analysis tail is wagging the specification dog”.

openSSL CORPORATION

TLS 1.3 brittleness

- TLS 1.2: a PSK was just (key, identity)
  - usable with any ciphersuite, any PRF hash
- TLS 1.3: (key, identity, hash)
  - the key schedule's first input, so it needs a hash
- RFC 9258: and the protocol version too
  - cross-version reuse may produce related outputs

The operational cost

`2 hashes × 2 versions`

= 4 PSK identities + binders to put in the extension

…for one shared secret.

"Just rotate the key" stops being a single action.

openSSL CORPORATION

Formal analysis costs

A pre-agreed digest (KDF) was not the only restriction in aid of formal analysis.

- **A resumption PSK may request client certificates** — post-handshake only (§4.4.2)
- **An external PSK may not** — not in the handshake, and not afterwards either
  - the binder transitively covers the server's certificate for a ticket; for an external PSK it does not (Appendix F.1.1)
- **Key schedule domain separation** — the binder key uses "ext binder" or "res binder"

Processing depends on how the identity was matched

openSSL CORPORATION

RFC 9258: the PSK importer

```
struct {
    opaque external_identity<1...2^16-1>;
    opaque context<0..2^16-1>;
    uint16 target_protocol;
    uint16 target_kdf;
} ImportedIdentity;

epskx = HKDF-Extract(0, epsk)
ipskx = HKDF-Expand-Label(epskx, "derived psk",
                          Hash(ImportedIdentity),
L)
```

epsk — the external key as provisioned
epskx — intermediate extracted entropy
ipskx — the imported PSK: L bytes, this identity

external_identity — what you provisioned

context — binds the deriving protocol

target_protocol — the TLS version

target_kdf — the digest

The identity says which protocol version and digest the key is for.

One provisioned key, one PSK per (KDF, protocol version).

The hash binding is not relaxed — multiple PSKs are derived instead. Offer them all; the server picks.

openSSL CORPORATION

External PSKs

benefits and requirements

What external PSKs are for

Replacing certificate authentication with proof of possession of a symmetric shared secret.

- **Bridging an existing authentication system** — e.g. a GSSAPI session key as the PSK
- **Devices with no PKI** — one key per client; knowing it proves which device you are
- **Keys from elsewhere** — QKD, but RFC 9973 (PSK + certificate) may be a prudent backstop
- **Anonymous TLS** — a published key: anyone can connect, nobody is authenticated
  - no implementation yet — what the last portion of the talk is about

What a PSK has to be

- **Known only to the peer(s) you expect** — PSKs replace the server and client certificates
  - whoever knows the secret can authenticate as the peer
  - for resumption that is natural — the ticket came from the server
  - or server cluster, with shared ticket decryption keys
- **Long enough to resist brute force** — the binder is a MAC an attacker can see

Too short, and it is easily recovered offline.

The server may ignore your PSKs

- **The server may decline every PSK you offer** — and do a certificate handshake instead
  - "if no acceptable PSKs are found, the server SHOULD perform a non-PSK handshake if possible"
  - "if possible" means both ends share a signature algorithm for which the server has a certificate
- **A client relying on PSK authentication must check one was used**
  - in OpenSSL today that is a manual check
  - **Or refuse certificates up front** — offer PSKs and omit `signature_algorithms`
  - the server must then use a PSK or fail the handshake
  - OpenSSL client does not support this yet; the server side already accepts it
- **With the RFC 9258 work** — the session will record a peer identity for PSK authentication too

You have to ask whether the PSK was used.

OpenSSL

PSK implementation features and issues

First, get the existing code right

- RFC 9258 adds a new way to derive PSKs. It does not fix the code that uses them.
- So the job became a review of the whole PSK implementation:
  - against what the RFCs actually require
  - against the problems already reported, and PRs in progress — subsumed and expanded

Seven commits later — none of them the feature I set out to add.

Three of them, polished and/or extended, but under the original contributor’s name.

Client certs, no Session ID context: every PSK failed

- **Fatal alert, not a fallback**
  - tickets and external PSKs alike (external PSKs now exempted)
- **A fresh handshake still succeeded** — minting resumption PSKs the same server would refuse
  - the failure happens on the connection that attempts resumption (poison ticket no longer issued)
- **Why the check exists** — partitions a shared resumption session cache
  - an external PSK carries no precomputed authentication result — it is validated on this connection
  - on the client, the reverse: a session ID context that was set rejected PSKs (now accepted)

It asked whether this was a PSK, not whether it was a resumption

0-RTT requirements

- **Cipher suite** — resumption needs only the same hash; 0-RTT needs the exact suite
  - early data now refused on a mismatch; resumption itself is unaffected
- **ALPN** — must match the protocol the PSK negotiated
  - wasn't checked at all — early data now refused, the handshake continues without it
  - a new ticket also kept the ALPN of the initial connection, not the one that issued it — now refreshed
- **Ticket age** — a ticket past its lifetime must not be offered
  - the client stopped sending the PSK but kept advertising early data — now both suppressed

Don't offer what you can't use

- **The client offered PSKs the server was bound to refuse**
  - a ticket is usable only if some offered ciphersuite uses its hash — otherwise retired, next candidate tried
- **And attempted 0-RTT that could not be accepted**
  - the first PSK's exact suite must be offered, and its ALPN among those offered
  - a client-detected ALPN mismatch used to be fatal; it now suppresses early data only
- **An application's PSK callback could hand back a session with no key in it**
  - the error was detected late, not at the callback — now rejected as a bad PSK

Duplicated possibly inconsistent checks

- **Building one ClientHello, the client decided the same things in more than one place**
  - is the ticket usable; which PSK goes first; is that PSK's cipher actually on the wire
- **The** `early_data` and `pre_shared_key` extensions were built separately — and could disagree
  - an application's security callback need not answer the same way twice
- **Early data set up against a ticket that was then dropped:**
  - the binder never ran, so the early secret was left all zero
  - and the 0-RTT data went out under keys a passive observer can rebuild from the ClientHello

Padding extension needed size of subsequent PSK extension (which must be last)

Encrypted Client Hello (ECH), supposed to mimmic inner PSKs (protocol design error)

Graceful handling of application (mis)behaviour

- **A server callback could return a session with no cipher, or a TLS 1.2 one**
  - the code read that cipher's handshake digest — a NULL dereference, reachable from the application
  - a TLS 1.2 cipher passed the pointer check and gave a meaningless digest
  - now skipped: the PSK is ignored and a full handshake proceeds
- **On the client, the old PSK callback's error paths leaked**
  - the raw secret left uncleansed on the stack, and on one path an SSL_SESSION never freed
  - now a single exit that cleanses the whole buffer and frees the session

Neither is a protocol bug. Both are reachable from correct-looking code.

Next steps

anonymous TLS, and knowing who you talked to

openSSL CORPORATION

Back to the few hours

- **A well-known external PSK, imported per RFC 9258, with no peer identity to assert**
  - `OSSL_TLS13_PSK_new_import_anon()` — anonymous is import-only
- **Usually one PSK. Two during a rollover — offer the old and the new**
  - the server takes the first identity it recognises, and validates only that binder
  - each side can adopt the new key independently
- **The library enforces the safety properties, not the application**
  - an anon PSK will not resume under `psk_ke` alone — forward secrecy is not quietly dropped
- **Configured in advance — no callback needed**
  - `SSL_CTX_add1_tls13_psk()`
  - s_client and s_server extended to support RFC9258 imported PSKs

PSK-based session identity

- **A TLS 1.3 PSK could not say who the peer was — or that nobody was**
- **The session now carries an opaque peer identity and a status**
  - `SSL_SESSION_set1_identity()` / `SSL_SESSION_get0_identity()`
- **Three status values, not two:**
  - `VALID` — authenticated. `NONE` — explicitly unauthenticated. `UNKNOWN` — never asserted.
  - an anonymous PSK asserts `NONE`: the library says “nobody”, not “don’t know”
  - callbacks written before this stay `UNKNOWN`, and keep resuming as they did

Connection verify_result updated accordingly

openSSL CORPORATION

Smaller pieces first

- **The PSK extension handlers were split into helpers before anything was added**
  - smaller functions, each with one decision in it — and one place to fix it
- **The ECH GREASE PSK goes away**
  - the outer ClientHello faked a pre_shared_key: random bytes shaped like the real one
  - every change to the real offer had to be mirrored in the fake
- **And the padding I was carefully recomputing — gone**
  - an unrelated cleanup deleted the extension — the bookkeeping went with it

Q&A

TLS 1.2

PSK as a ciphersuite family

TLS 1.2: PSK ciphers

`TLS_PSK_WITH_AES_128_CBC_SHA`	pre-shared key only
`TLS_DHE_PSK_WITH_AES_128_GCM_SHA256`	+ forward secrecy
`TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256`	+ forward secrecy, elliptic curve
`TLS_RSA_PSK_WITH_AES_256_GCM_SHA384`	+ a server certificate as well

The same pre-shared key works with all of them.

Enable one or more `aPSK:kRSAPSK:!eNULL` ciphers in desired preference order.

RFC 4279 · RFC 5487 · RFC 5489

TLS 1.2: resumption not PSK based

- **Session IDs** — the server caches the session; the client sends the ID back (RFC 5246)
- **Session tickets** — the server encrypts the session to itself; the client stores it (RFC 5077)
- Either way: an abbreviated handshake reusing the previous master secret
- Unlike TLS 1.3, **no asymmetric key exchange as part of resumption**

Resumption was not a PSK — two mechanisms, two sets of rules.

TLS 1.3 merges the two into one mechanism.

TLS 1.2: PSK API (callback)

```
unsigned int (*SSL_psk_client_cb_func)(SSL *ssl, const char *hint,
        char *identity,     unsigned int max_identity_len,
        unsigned char *psk, unsigned int max_psk_len);

unsigned int (*SSL_psk_server_cb_func)(SSL *ssl, const char *identity,
        unsigned char *psk, unsigned int max_psk_len);

/* Optional server hint to client, application-specific */
int SSL_CTX_use_psk_identity_hint(SSL_CTX *ctx, const char *hint);
```

Return identity string and key bytes.

No cipher. No protocol version. No digest. The key does not know about any of them.

What's left

- **Twelve commits on a branch, targeting OpenSSL 4.2**
- **Postfix support written and tested — not yet buildable against any release**
  - Calls APIs that exists only in my worktree
- **The IETF draft comes last, deliberately**
  - describe something that already interoperates, not something that might

Still not a few hours.