TLS 1.3

`draft-ietf-tls-tls13-12`

Eric Rescorla

Mozilla

ekr@rtfm.com

Overview

- Changes since draft-10
- Outstanding consensus calls
- 1-RTT PSK and session tickets
- Context values
- Key schedule and key separation
- 0-RTT details
- Minor issues

Changes since draft-10

- Restructure authentication along uniform lines *
- Restructure 0-RTT record layer *
- Reset sequence numbers on key changes
- Import CFRG Curves
- Zero-length additional data for AEAD
- Revised signature algorithm negotiation *
- Define exporters *
- Add anti-downgrade mechanism *
- Add PSK cipher suites
- Other editorial

Restructuring Authentication

- TLS 1.3 has four authentication contexts
  - 1-RTT server
  - 1-RTT client
  - 0-RTT client†
  - Post-handshake
- All were slightly different
- draft-12 unifies them into one common idiom

† Marked for death.

TLS 1.3 Authentication Block

- Three messages: Certificate*, CertificateVerify*, Finished
- Inputs
  - Handshake Context (generally the handshake hash)
  - Certificate/signing key
  - Base key for MAC key
- CertificateVerify =

  `digitally-sign(Hash(Handshake Context + Certificate))`*
- Finished =

  `HMAC(finished_key, Handshake Context + Certificate + CertVerify)`
- Different finished keys for each direction (based on Base Key)

---

*Includes disambiguating context string.

Eye Chart

Mode	Handshake Context	Base Key
0-RTT	ClientHello + ServerConfiguration + Server Certificate + CertificateRequest (where ServerConfiguration, etc. are from the previous handshake)	xSS
1-RTT (Server)	ClientHello ... ServerConfiguration	master secret
1-RTT (Client)	ClientHello ... ServerFinished	master secret
Post-Handshake	ClientHello ... ClientFinished + CertificateRequest	master secret

Restructure 0-RTT Record Structure

- draft-10 had a somewhat idiosyncratic design
- draft-12 0-RTT parallels 1-RTT
  - `handshake` for handshake data
  - `application_data` for application data
  - New `end_of_early_data` (warning) alert for separation
  - Separate handshake and traffic keys

Revised Signature Algorithm Negotiation (I) (davidben)

- TLS 1.2:

```
struct {
      HashAlgorithm hash;
      SignatureAlgorithm signature;
} SignatureAndHashAlgorithm;
```

- Curves were orthogonal (`supported_curves`)
- It seemed like a good idea at the time
- ... but new signatures algorithms are tied to one hash for each curve size
- Proposal from davidben: define a new structure that ties everything together

Revised Signature Algorithm Negotiation (II)

```
enum {
    // RSASSA-PKCS-v1_5 algorithms.
    rsa_pkcs1_sha1 (0x0201),
    rsa_pkcs1_sha256 (0x0401),
    rsa_pkcs1_sha384 (0x0501),
    rsa_pkcs1_sha512 (0x0601),

    ...
} SignatureScheme;
```

- These line up with the existing code points
- New code points define the triplet: signature algorithm, curve, hash

Define Exporters

- RFC 5705 defined exporters in terms of the PRF
  - We removed the PRF....
- New definition:

```
HKDF-Expand-Label(HKDF-Extract(0, exporter_secret),
                  label, context_value, length)
```

- Note: this doesn't cover 0-RTT. More on this later.

Anti-Downgrade Mechanism I (Green/Bhargavan)

- TLS 1.2 and below downgrade defense was tied to the Finished message
- TLS 1.3 downgrade is tied to both Finished and server CertificateVerify
  - So TLS 1.3 resists downgrade even when the key exchange is weak
  - ... but what about downgrade to TLS 1.2 or 1.1

Anti-Downgrade Mechanism II (Green/Bhargavan)

- Countermeasure: taint the `ServerRandom`
  - If server supports TLS 1.2 or TLS 1.3 but client offers a lower version use a special `ServerRandom`
    - Top eight bytes are `44 4F 57 4E 47 52 44 01` (TLS 1.3) or `44 4F 57 4E 47 52 44 00`
    - This is covered by the server signature
  - Clients MUST check
- This doesn't protect you if you negotiate to static RSA
  - Didn't you want PFS anyway

Mailing List Recap: 0-RTT Client Authentication

- Current design: client signs the `ClientHello+...<Server context>`
  - The authentication is tied to the client's (EC)DH share
- This is very brittle
  - Effectively it's a long-term DH certificate
    - Modulo anti-replay issues
  - Compromise of either DH share allows impersonation
- 0-RTT PSK also scary
- Proposal on list: Remove 0-RTT Client Authentication entirely

(EC)DHE-based 0-RTT

- Currently we have 0-RTT modes
  - (EC)DHE: Server provides (EC)DHE static key in ServerConfiguration and pairs it with its ephemeral
  - PSK: Based on session ticket
- Proposal: only do the PSK-based mode (Fournet et al., Sullivan et al.)
  - People are going to want to do PSK-resumption anyway for perf reasons
  - Implicit binding between connection parameters
  - No need for a ServerConfiguration object
  - The crypto analysis of (EC)DHE 0-RTT is tricky
  - Can always re-phrase DH as a "PSK type" later

Objection: What about out-of-band priming?

- You can publish an (EC)DH key (e.g., in the DNS)
  - 0RTT-PSK isn't compatible with out-of-band priming (duh!)
- But...
  - This brings in all the concerns about delegation
  - No really plausible priming mechanism (DNS not viable)
  - See previous comments about DH-as-PSK

Objection: Security impact of client-side storage

- Storing a DH public key requires only storage integrity
- Storing a PSK requires secrecy
- But...
  - Client-side secure storage already needed for session caching
  - Generally session caches don't survive program shutdown
  - Google's measurements in QUIC show this has no performance impact versus long-term storage

Objection: PFS

- With (EC)DHE you get
  - No PFS for 0-RTT data
  - PFS for 1-RTT data
- Can do PSK 0-RTT two ways
  - PSK only (no PFS)
  - PSK-(EC)DHE (same PFS as with DH 0-RTT)
- Note: can do better with server-side state as opposed to tickets

Objection: WebRTC

- WebRTC might have a use for this
- But...
  - We have a different hack for that (`draft-rescorla-dtls-in-udp`)

Objection: Server Proof of Private Key

- The DHE 0-RTT mode forces the server to re-sign every time
  - The point of PSK is to avoid the server doing that
- This creates a tradeoff between 0-RTT and continuing proof of server key
- Solution: Allow 0-RTT PSK to be used with signed (EC)DHE exchange*

---

*Details TBD.

Proposal: Remove 0-RTT DHE-based mode

- The only 0-RTT mode will be PSK
- We can re-add 0-RTT DH mode later if needed
  - Probably more oriented towards external priming

NewSessionTicket Format (Bhargavan)

- NewSessionTicket just has expiry.... more information needed
  - Cipher suites the server would accept (ECHDE-PSK or PSK, especially)
  - Which 0-RTT modes you would accept: None, Replayable, All (????)

```
enum {
        no_early_data_allowed(0),
        replayable_early_data_allowed (1),
        all_early_data_allowed(2),
        (65535)
 } EarlyDataType;

 uint32 ticket_lifetime;
       opaque ticket<0..2^16-1>;
       CipherSuite cipher_suites<2..2^16-2>;
       EarlyDataType early_data_type
 } NewSessionTicket;
```

0-RTT PSK Extensions I

- We do need extensions to contextualize 0-RTT data
  - ALPN
  - Elapsed time (PR#437)
- Where do they go?
  - `EarlyDataIndication.extensions`
  - `EncryptedExtensions` (let's add this back)
- Relationship to original connection?

0-RTT PSK Extensions II: Where do they go?

- `EarlyDataIndication` has an extensions field
  - But this is in the clear
  - As much stuff as possible should be secret
- We have gone back and forth on client `EncryptedExtensions`
  - We should add it back
  - Minimally want it for privacy-leaking data like elapsed time
  - Semantics: *only* apply to the 0-RTT data
- Proposed dividing line: same as for `ServerHello.extensions/EncryptedExtensions`

0-RTT PSK Extensions III: Semantics

- Two basic options
  - Omit all the extensions and require both sides to use what was picked last time
  - Client sends the relevant extensions (defining what it expects the server to want) and the server can reject if it choked
- "Matching" options
  - Extensions must match the 1-RTT negotiation (Requires both sides to keep the same configuration)
  - Extensions must match the last negotiation (Requires both sides to remember)
- Proposal: extensions MUST be the same as last time and server must reject 0-RTT if its config changes

Rejection of 0-RTT: HelloRetryRequest (Bhargavan)

- Setting: client offers PSK with 0-RTT
- ... server sends `HelloRetryRequest`
- What happens to the 0-RTT data
  - Can it be resent on the next flight
- Proposal: No. `HelloRetryRequest` sends you back to the beginning.

Rejection of 0-RTT: Finding the next handshake block

- What happens if server rejects 0-RTT?
- Need to skip ahead to next non 0-RTT client message
  - `HelloRetryRequest` $\rightarrow$ wait for `ClientHello`
  - `ServerHello` $\rightarrow$ wait for `Certificate` or `Finished`
- Right now this means trial-decryption
- Karthik suggests that the client sends `end_of_early_data` alert in the clear upon rejection
  - Probably easier to implement, very slightly worse privacy
- Proposal: Adopt this

0-RTT Exporters

- We haven't defined any
- We need them
  - For Tokbind
  - For QUIC
- MT will be a sad panda

- Construction needed...

End of Day 1

Key Separation: A Layman's View

- Basic idea: different keys for different purposes
  - For example, handshake and application data
- Why? Analyze different pieces separately
  - ... and then put them together
- Handshake: establish parameters and output traffic keys
- Application Layer: take traffic keys and protect traffic
- If you use separate keys, handshake doesn't depend on application layer security
  - And to some extent vice versa, as long as handshake delivers on certain guarantees

TLS Key Separation Issues

- TLS 1.2 used the same keys to encrypt handshake and application data
  - Specifically, `Finished` message
  - This can still be proven secure but its far more difficult
- TLS 1.3 generally has different handshake and application keys
- Exceptional cases
  - `NewSessionTicket`
  - Post-handshake authentication
  - `KeyUpdate`
- Also, 0.5RTT vs. 1RTT data

What is 0.5 RTT Data?

Client | Server

ClientHello [Random, g^c]

ServerHello [Random, g^s]...Finished

Application data ("0.5RTT")

Finished

Application data ("1RTT")

With client auth?

Client | Server

ClientHello [Random, g^c] →

ServerHello [Random, g^s]...Finished ←

Application data (to anonymous user) ←

Certificate...Finished →

Application data (to authenticated user) ↔

Non-digression: Retroactive authentication

- Data originally interpreted as an anonymous peer
- Then you authenticated
  - Now reinterpreted as an authenticated peer
- We have bad models for this
  - But it happens all the time (e.g., shopping carts)
- Application semantic even if we have a cryptographic separation

One more thing about 0.5 RTT Data

- The server sends it before the client proves its live
- If you're using PSK, this means that attackers can get the server to replay
- Like a weaker version of 0-RTT replay issue

Possible Resolutions

1. No change
2. Warn against/forbid 0.5 RTT data when client auth is used
    - Possibly relax this if we get analysis that it is safe
3. Include client's second flight in 1RTT application keys
    - So you can't do 0.5 RTT with client auth
4. Change keys between 0.5RTT and 1RTT

- Proposal: #2.

Key Separation: Post-handshake Messages

- We have separated handshake and application data keys
  - ... but only for the main handshake
- Post-handshake messages that you might think of as handshake
  - `NewSessionTicket`
  - Client authentication
  - `KeyUpdate`
- This makes cryptographers sad
  - Because compromise of application keys might affect handshake

Demuxing Options

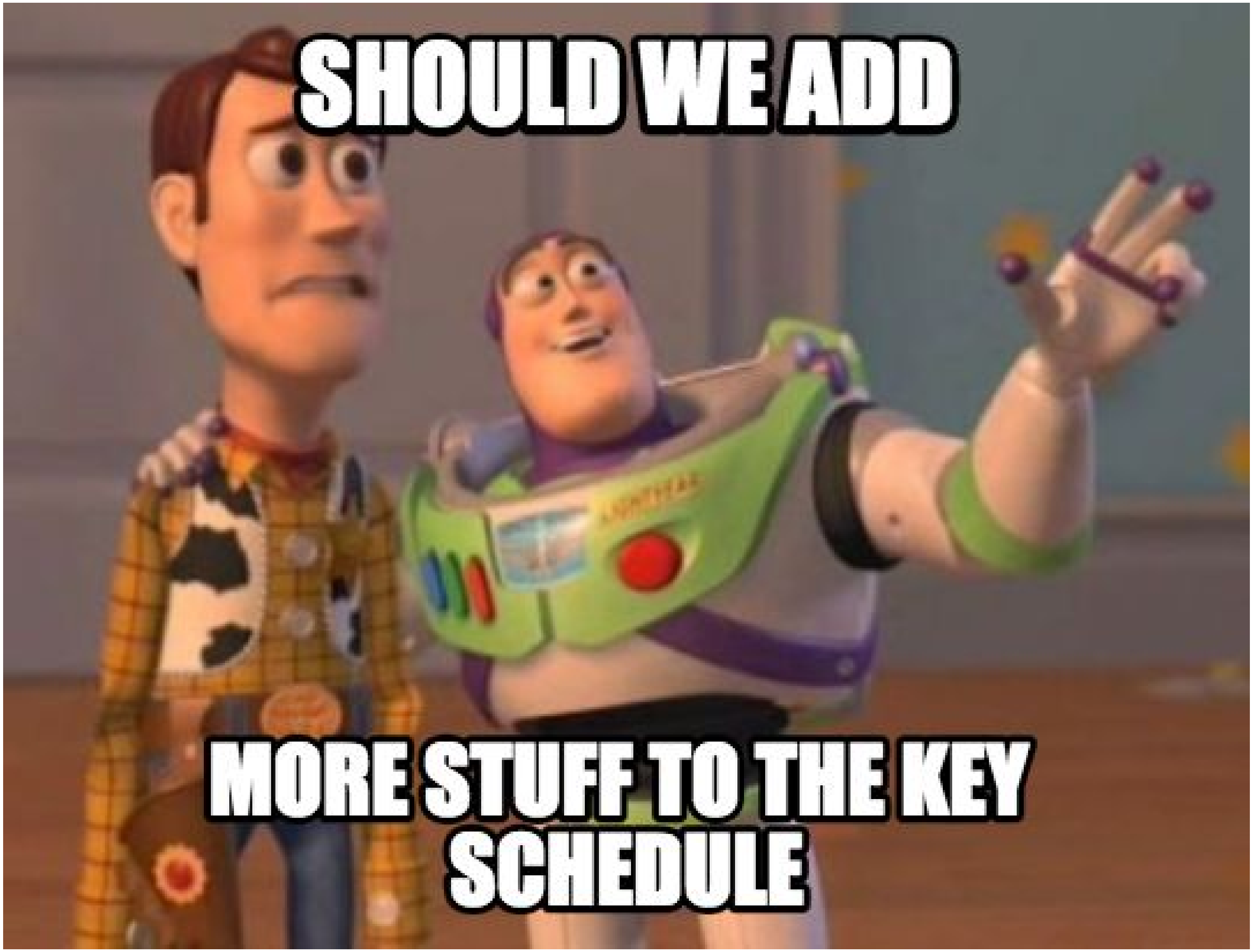
- Two keys in use concurrently
  - Handshake (or post-handshake)
  - Application
  - First time this happens in TLS
- How do I know which key is being used?
  - Trial decryption
  - ~~Wrap handshake-encrypted messages in application keys~~
  - Restore the content type byte
- Based on Tuesday, trial decryption seems best (if we do this at all)

What would be encrypted under handshake keys?

`NewSessionTicket`	Yes
Client Authentication	Yes
KeyUpdate	???
Alerts	No

Which key?

- Existing handshake traffic key
- New post-handshake traffic key
- Minor additional complexity in key schedule


SHOULD WE ADD
MORE STUFF TO THE KEY
SCHEDULE

Key Context (yes, yes, more context)

- Life has gotten simpler since we got rid of DHE 0-RTT
  - But the whole question of context seems a little brittle
  - cf. the Scott et al. paper from last year
- Karthik proposed being more explicit about binding context into the handshake
  - This would strengthen a bunch of stuff

What do we mean by context?

- PSK/Resumption-PSK: Some public function of the key
  - E.g., `HKDF(PSK, <fixed label>)`
- DHE 0-RTT (if we bring it back): the server context
  -

    `ServerConfiguration + ServerCertificate + CertificateRequest`

Explicit Binding

```
struct {
        opaque psk_identity<0..2^16-1>;
        opaque context<0..255>;
 } PreSharedKeyInfo;

 struct {
        select (Role) {
            case client:
                PreSharedKeyInfo keys<2..2^16-1>;

            case server:
                uint16 index;  // The selected index
        }
  } PreSharedKeyExtension;
```

- Client supplies the context value in `ClientHello`
- Server checks it (important!)
- Automatically included in the handshake hash

Implicit Binding

- One *unreviewed* possibility*

```
Option 1: Include in SS
     K_hh = HKDF-Expand-Label(xSS, Handshake Hash Key, )
     handshake_hashes = HMAC(K_hh, Hash(Handshake messages))
     // IMPORTANT: Need to revise SS if we re-add DHE-0-RTT

Option 2: Use directly
     handshake_hashes = Hash(Hash(Context) || Hash(Handshake messages))
```

- Every time we use handshake hashes mix in something derived from context
- Client and server implicitly do this (no new signaling)

---

*Warning, potentially busted handwaving.

Simplified Key Schedule

- The current key schedule is agnostic about the order when we get SS and ES
  - But for all known modes we get SS (if at all) then ES (if at all)
- This suggests a simpler (linear) key schedule

```
             0
             |
      SS -> HKDF
             |  \
             |   \
             v    v
             X1   0-RTT Traffic Keys *
             |
             |
             v
      ES -> HKDF [ClientHello, ServerHello]
             |  \
             |   \
             v    v
             X2    1-RTT Traffic Keys *
             |
             |
             v
       0 -> HKDF [ClientHello...ClientFinished]
             |
             |
             v
             RMS, EMS
```

Issue #215: Let servers send known groups

- Right now client sends some set of keys
  - P-256, X25519, etc.
- Server picks one
- No way for server to tell client "I would take group A, but I would prefer/would also take group B"
  - Without rejecting (ugh!)
- Easy fix: allow server to send `SupportedGroups` in `ServerHello`

Issue #426: Receive Generation field in `KeyUpdate`

- Some people want to build TLS monitoring systems that aren't MITM
- Idea: update traffic keys to generation $N+1$, then release keys N to monitoring device
- Issue: how do you have partially trusted devices?
  - That can't inject traffic
  - Client knows when it has updated its receive key but not when the server has
- Proposed fix: add a "receive generation" field to `KeyUpdate` so client knows when it is safe.


I CAN'T BELIEVE
WE'RE STILL TALKING ABOUT
PSS

Implementation Status

Name	Language	ECDHE	DHE	PSK	0-RTT
NSS	C	Yes	No	Yes	Yes*
Mint	Go	Yes	Yes	Yes	Yes
nqsb	OCaml	No	Yes	Yes	No
ProtoTLS	JavaScript	Yes	Yes	Yes	Yes
miTLS	F*	Yes	Yes	Yes	???

- NSS interops with Mint and ProtoTLS
  - NSS 0-RTT in unintegrated branch
- ProtoTLS interops with nqsb
- Other combinations untested