We need an rfc9147bis

13 DTLS 1.3 DeTaiLS

David Benjamin

1. KeyUpdate problems

TLS 1.3 KeyUpdates apply immediately — stream is ordered

DTLS 1.3 KeyUpdates wait for a peer ACK

> “[...] implementations MUST NOT send records with the new keys [...] KeyUpdate has been acknowledged.”

ACKing KeyUpdate != process KeyUpdate!

Other post-handshake messages may be there

1. KeyUpdate problems

If 10 is lost, server changes keys too early

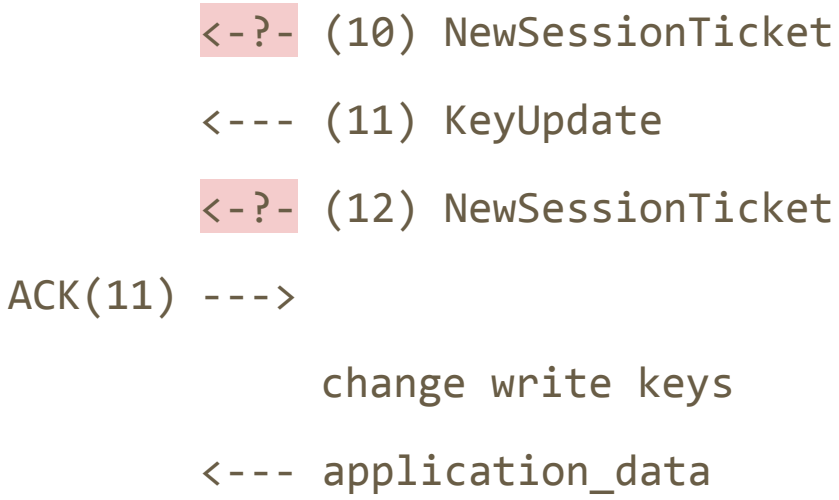
Client will not change keys yet:

> “DTLS implementations maintain [...] a next_receive_seq counter. [...] if [a message] matches next_receive_seq, next_receive_seq is incremented and the message is processed. [...] If the sequence number is greater than next_receive_seq, the implementation SHOULD queue the message [...].”

If 11 is lost, cannot retransmit 12!

> “Implementations MUST [retransmit] using the same epoch and keying material as the original transmission.”

Client does not even know 12 exists! Will drop old epoch.



1. KeyUpdate problems

Fix sender behavior:

- Change keys when KeyUpdate *and all previous messages* are ACKed
- No new handshake messages allowed in epoch after KeyUpdate (defer them to KeyUpdate ACK)

(Other options discussed on list)

https://www.rfc-editor.org/errata/eid8047

https://mailarchive.ietf.org/arch/msg/tls/_ku3-YDcroNmG_QKZsYTtqYzC0M/

2. ACKs near the version transition

DTLS transitions from unknown version to known version

ACKs are 1.3-only. How do ACKs work before version is known?

- Server receives part of ClientHello — no one knows version
- Client receives part of ServerHello — server knows version, client does not!
- Must also consider packet reordering!

Diagram in Section 6 suggests a client that sends ACK before ServerHello... but client does not know the version yet!

https://mailarchive.ietf.org/arch/msg/tls/ZEj04LyL3hJXeK1nsiOBoB2vCsg/

3. Epoch management is unspecified

A DTLS 1.3 implementation juggles many epochs

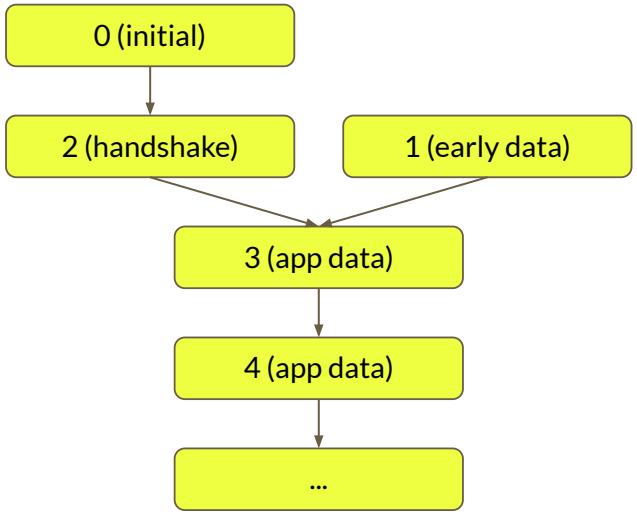
- Epoch 1 (early data) is weird
  - Section 7 says to ACK at highest epoch, but peer may not read epoch 1 ACKs
  - 0-RTT reject + HelloRetryRequest => rewind from epoch 1 to epoch 0 (initial)
  - Activating epoch 2 (handshake) should not close epoch 1
- After sending KeyUpdate ACK, MUST keep old and new read epochs
- Also true in handshake??
  - Yes. Peer might send ACKs at earlier epochs if, e.g., part of ServerHello is lost
- What does “current epoch” even mean?

Section 4.2 does not spell this out, some rules missing altogether

3. Epoch management is unspecified

We need to write this more clearly. Best guess:

- Epochs form a partially ordered set 😭
- Discard write epochs if not latest *and* no unacked messages use it
- New read epochs only become “current” when you receive a record from them
  - ….except epoch 1 which is never “current” for handshakes. Epoch 1 is out of sequence, until it isn’t.
  - Pending read epoch => no new handshake messages
- Past read epochs may be retained for reordering
  - If you do this, be very careful about what you allow in past epochs



* Actually these rules do not *quite* handle an unimportant edge case *ideally*

Intermission: 0.5-RTT KeyUpdates are hilarious

0.5-RTT KeyUpdates… are they a thing?

```
ClientHello       --->
                  <--- ServerHello, {EE..Finished}
                  <--- [application data]
                  <--- [KeyUpdate]
{Finished}        -/->
[ACK(KeyUpdate)] ---> (server can’t decrypt this)
```

Server epoch 3 does *not* implicitly ACK client Finished!

Client must keep retransmitting Finished

Infinite 0.5-RTT KeyUpdates??

```
ClientHello        --->
                   <--- SH{EE..Fin}
                   <--- [KeyUpdate1]
{Finished}         -/->
[ACK(KeyUpdate1)] --->
                   <--- [KeyUpdate2]
[ACK(KeyUpdate2)] --->
                   <--- [KeyUpdate3]
[ACK(KeyUpdate3)] --->
                   ...
```

Can we keep going?

No

Server cannot read ACK(KeyUpdate1),

so it cannot send KeyUpdate2.

Except...

Phantom 0.5-RTT KeyUpdates

```
ClientHello          --->
                     <--- SH{EE..Fin}
                     <--- [KeyUpdate1]
{Finished}           --->
                     <-/- [ACK(Finished)]
[ACK(KeyUpdate1)]    --->
                     <--- [KeyUpdate2]
[ACK(KeyUpdate2)]    --->
                     <--- [KeyUpdate3]
[ACK(KeyUpdate3)]    --->
                     ....
```

What if ACK(Finished) was lost?

Client *can* infer from epoch 4+ that server got Finished…
…but this is not a listed implicit ACK condition in Section 7.1!

So, currently, yes, you must keep retransmitting Finished until that ACK gets through

Back to our regularly scheduled programming

4. 0-RTT

Not all TLS/DTLS 0-RTT differences are specified

- Should override RFC 8446 Appendix D.3 — DTLS 1.2 servers *can* skip DTLS 1.3 early data... but beware multi-record datagrams
- Server receives half of client Finished — read epoch 1 (early data) should still be active
- After handshake, server may retain epoch 1 for reordering... what are the replay consequences? Must this be surfaced to calling application?
- Import some text from RFC 9001, Section 5.7?
  https://mailarchive.ietf.org/arch/msg/tls/tiMHeccRAT4DlPM_NRViPPoUYGs/

5. Post-handshake messages are not well-defined

> “Messages of each category can be sent independently, and reliability is established via independent state machines, each of which behaves as described in Section 5.8.1.”

Scope of duplication is ambiguous. State machines are inherently connected

Same message_seq namespace — still process in order?

Sender *may* simply by only allowing one unACKed outgoing flight — not written down

- Adds some deadlock risks that must be considered
- Has the complex case even been exercised?

ACKs may span post-handshake transactions, but still feed separate state machines?

- Which transactions do you retransmit?
- Separate but interacting timers? How?

Multi-message flights (Cert/CertVerify/Finished) must be consecutive sequence numbers

References to “the current flight” are ambiguous or wrong

6. ACKs and flights

Section 7 says to sort record numbers numerically (not well-defined)

- Does anyone do this?

Section 7 says to clear ACK queue when receiving *start* of flight N+1

- Should be when *sending* flight N because that ACKs N-1

Still not sufficient to ensure ACKs are single-flight

> "When an implementation detects a disruption in the receipt of the **current incoming flight**, it SHOULD generate an ACK that covers the messages from **that flight** which it has received and processed so far."

Post-handshake further complicates this

> "For post-handshake messages, ACKs SHOULD be sent once for each received and processed handshake record (potentially subject to some delay) and **MAY cover more than one flight**. This includes records containing messages which are discarded because a previous copy has been received."

https://mailarchive.ietf.org/arch/msg/tls/kjJnquJOVaWxu5hUCmNzB35eqY0/

7. HelloVerifyRequest

HelloVerifyRequest is forbidden in DTLS 1.3, but arrives *before* version is learned

What to do with the PSK binder when responding to HelloVerifyRequest?

RFC 6347 implies recomputing it is OK:

> “When responding to a HelloVerifyRequest, the client MUST use the same parameter values (version, random, session_id, cipher_suites, compression_method) as it did in the original ClientHello”

But list discussion suggests some stacks check extensions too

https://mailarchive.ietf.org/arch/msg/tls/HSTgV2voS9tn03oasGmBAdwazWA/

8-13. Miscellaneous

8. Forbid message_seq overflow https://www.rfc-editor.org/errata/eid8048

9. 64-bit epoch numbers were a waste; limited to 16-bit in practice: https://www.rfc-editor.org/errata/eid8050

10. Forbidding compatibility mode is subtle: https://www.rfc-editor.org/errata/eid8066

11. No guidance on sending short sequence numbers: https://mailarchive.ietf.org/arch/msg/tls/3IEIzc2ssdkZbyYEXlvz6t3ZhbI/

12. Shouldn't accept ACKs at the wrong epoch: https://www.rfc-editor.org/errata/eid8108

    [NB: erratum should have an erratum]

13. Grouping things to keep the count at thirteen

    https://www.rfc-editor.org/errata/eid8141

    https://www.rfc-editor.org/errata/eid8051

    https://www.rfc-editor.org/errata/eid8100

Next steps

We need an rfc9147bis

Involve existing DTLS 1.3 implementers:

- Retain compatibility
- Find out how much was ever implemented at all