Time to take stock

The completeness of an implementation of TLS1.3 is measured by the number of extensions it supports. So lets list them, see where we stand, and work out some priorities. Its worth noting that no implementation (not even OpenSSL!) implements everything. Note that a client, depending on its circumstances, may use only a subset of the available extensions.

Most extensions can appear in more than one place – typically a request from one party, followed by an acknowledgement from the other. Extensions can appear in the Client Hello (CH), the Server Hello (SH), a Certificate Request (CR), the Certificate itself (CT), in a Handshake Retry Request (HR), in a session ticket (ST) or in the server's Encrypted Extensions (EE). As always for more information refer to https://www.davidwong.fr/tls13/

1. Server Name (CH, EE)

Fully implemented

2. Max Fragment length (CH, EE)

Fully implemented. Useful way for a small device to defend itself against being overwhelmed by big chunks of data. However Servers may choose to ignore it.

3. Status Request (CH, CR, CT)

Not implemented. This is OCSP stapling. Briefly the idea is that the client, who is unwilling or unable to check certificate revocation status, requests further evidence from the server that its certificate is kosher, and gets the server to do the heavy lifting This requires a fresh digital signature from the Certificate Authority (CA) "stapled" onto the certificate which confirms that it has not been recently revoked by that CA. The server response is actually embedded as a certificate extension. See below.

4. Supported groups (CH, EE)

Fully implemented, as indeed it is mandatory.

5. Signature Algorithms (CH, CR)

Fully implemented. Mandatory for full CH, but also required in CR in case client is also authenticating with a certificate, in response to a request from the server. Not required for CH on session resumption (which does not require a certificate chain).

6. Use Secure Real-Time Transport Protocol (srtp) (CH, EE)

Not implemented. Applies only to DTLS, and we are not supporting that (yet).

7. HeartBeat (CH, EE)

Not currently implemented, but simple to do. A mechanism to keep a connection alive.

8. Application Layer protocol negotiation (CH, EE)

Implemented. A mechanism for letting TLS know what kind of application is going to run over the connection once it is secured by TLS. For example a server might have different certificates for different applications. Currently HTTP is the only application supported.

9. Signed Certificate Timestamp (CH, CR, CT)

Not implemented. This is the mechanism commonly referred to as “Certificate Transparency”. It provides some confidence that the CA itself has not misbehaved. See below.

10. Client Certificate Type (CH, EE)

Not implemented. Allows raw public keys to be used by the client instead of self-signed certificates, hence keeping the size down. It may be a more elegant solution in the IoT setting, and can apply in a non-PKI setting, where some other mechanism is used to authenticate such keys. Even OpenSSL doesn’t support this one, but I think maybe we should. Since PQ public keys are bigger, this extension may help mitigate the impact of that.

10. Server Certificate Type (CH, EE)

Same as above, this time for the server.

11. Padding (CH)

Fully implemented. Random padding bytes are added to frustrate traffic analysis.

12. Key Share (CH, SH, HR)

Fully implemented, mandatory.

13. Pre-shared Key (CH, SH)

Fully implemented. A pre-shared key can be used to avoid certificates. It can either be derived from a “ticket”, in which case it is used for session resumption, or it can be something negotiated “out-of-band”

14. PSK Key Exchange Modes (CH)

Implemented, but only for the safer (EC)DHE mode (supports forward secrecy), not PSK alone.

15. Early Data (CH, EE, ST)

Fully implemented. Allows application data to be sent from client to server in the first flow. Can only happen on a session resumption, or with a pre-shared key.

16. Cookie (CH, HRR)

Fully implemented.

17. Supported Versions (CH, SH, HR)

Fully implemented. Only version 1.3 of TLS is supported.

18. Certificate Authorities (CH,CR)

Not implemented. Supplies a list of acceptable Certificate Authorities to the other party, and thus can be used to limit the size of the CA certificates that need to be stored.

19. OID filters (CR)

Not implemented. Pretty obscure.

20. Post Handshake Authentication (CH)

Not implemented. Again pretty obscure. Its a message from the client to the server indicating a willingness to authenticate at any time (by supplying its own certificate) post-handshake. Why not just do it during the handshake?

21. Signature Algorithms Cert (CH, CR)

Fully implemented. The requester of a certificate is indicating which types of signature on certificates it can verify.

Most of the currently missing extensions are related to more complete X.509/PKI support. A policy decision is the extent to which we want to get drawn down that X.509/PKI rabbit hole. Checking certificate chains plus providing full support for OCSP and Certificate transparency is a big resource intensive undertaking. At the moment we just have bare-bones support for this – do we really want to take it further? Maybe a client just needs to pass the whole certificate chain on to a trusted server (based on OpenSSL?) which does the checks and comes back with the Yes or No answer. Maybe some application specific light-weight solution can be found. For TLS we just need to be sure that the server's public key is genuine.

Another idea would be to exploit Certificate Pinning. The idea is simple – burn acceptable server certs into a client store, and only only accept certificates that are found in that store.

Which brings me back to the Trust Abstraction Layer idea, where our current light-weight X509/PKI is just one such solution. If anyone else wants to do the full-blown thing, be our guest, and just plug it in.

Research idea

Big problem with post-quantum TLS is large size of public keys and signatures. Now by using Identity Based Cryptography (instead of PKI) public keys can be eliminated completely. And by using KEMs instead of signature, signatures can be eliminated. See

https://dspace.mit.edu/bitstream/handle/1721.1/130100/Paper_PrePrint_Version.pdf?sequence=1&isAllowed=y

I have implemented the Ducas et al NTRU-based IBE scheme, it is very practical. Maybe IBE's time has finally come? If the server uses IBE, the client does not need its public key, which removes the extra round that KEMTLS required(?) Using IBE also allows a much simpler solution to the problem of encrypting the Client Hello - https://datatracker.ietf.org/doc/draft-ietf-tls-esni/

This paper entitled "The viability of Post Quantum X.509 Certificates" (https://eprint.iacr.org/2018/063) is also of interest.