What's in and what's out

These recommendations are at this stage for discussion purposes. No hard decisions have been made. Criteria for recommendations depend on

(I) Aiming for the AES-128 level of security, and not falling below this.
(II) remaining IoT friendly (that is support for constrained environments)
(III) not supporting little-used legacy methods
(IV) fast, efficient, minimum number of exchanges.

Recall that a client will typically issue the same "Hello" to every server, which dictates and constrains the server response. We could vary the client Hello depending on which server we are talking to, but right now I don't see the point in doing that – we will always be selecting the same choices predicated on the above criteria.

Key Exchange Modes

TLS supports three key exchange modes

1. (EC)DHE – Diffie-Hellman over finite fields or elliptic curves
2. PSK-only
3. PSK with (EC)DHE

I am not sure why finite fields are in there at all. No-one seems to support them, indeed I have seen their use deprecated. So recommend we do NOT support them (III). I am unsure as to the use case for PSK-only, it is clearly more secure to use PSK in conjunction with ECDHE (forward secure), so recommend we do NOT support PSK-only (I).

For ECDHE there needs to be a choice of elliptic curve. The client may send a single key share using its favourite curve, or it may send more than one, each one associated with a different curve. In the unlikely event that the server does not support any of these curves the server may issue an HelloRetryRequest, letting the client know which curve it does support. We will support HelloRetryRequest, but since it introduces a significant delay into the overall exchange, obviously we would prefer it not to be needed. The RFC tells us we MUST support elliptic curve secp256r1, and SHOULD support X25519. Other acceptable key-exchange curves are secp384r1, secp521r1 and X448.

For speed X25519 is probably the best choice, and in practise it should always be supported. So recommend using X25519 only in the clientHello. However we need to remain flexible here as we may want to offer a non-standard PQ key share as well, and if picked up by one of our own servers the PQ choice may be the preferred one. Or a hybrid X25519+PQ key exchange.

In the unlikely event that the server does not support X25519, we recommend support for secp256r1 as well, which ensures that the protocol can proceed to the next stage, although an HelloRetryRequest may be needed for this fall-back.

Since support for secp384r1 and secp521r1 will be required for digital signature (see below), these modes are also recommend to be supported (but probably not commonly used).

Cipher Suites

After exchanging Hellos the rest of the protocol is encrypted using a symmetric cipher-suite. Here we are constrained by the RFC to just 3 choices – AES128+SHA256, AES256+SHA384 and CHACHA20+POLY1305+SHA256. Only the first of these MUST be implemented. In this case if we offer a choice in clientHello, the server decides which one to use. Recommend supporting all three modes – the default AES128+SHA256 cipher-suite, AES256+SHA384 (for the PQ world), and CHACHA20+POLY1305+SHA256 as it is likely faster on smaller devices.

Signature Algorithms

The protocol transcript is digitally signed by the Server using the private key associated with the public key embedded in its X.509 certificate, which is sent to the client along with a supporting certificate chain leading back to a recognized Certificate Authority.

Here there are many more choices, as different X.509 certificates may support public/private key pairs associated with a variety of public key algorithms and elliptic curves. We do not have to support every possible algorithm – in which case we simply reject non-compliant certificates. As a policy we reject all certificates that fall below the AES-128 threshold (that means for example anything using SHA1).

Recommend we support RSA-2048 and RSA-4096 algorithms, with either PSS or (with some reluctance) PKCS1.5 padding.

Note that our Key Share recommendation will already pull in the code for X25519 and secp256r1, so we loose nothing by supporting ECDSA for secp256r1, and EdDSA for X25519. It is also recommended to support ECDSA for secp384r1 and secp521r1 as these curves are commonly used by X.509 certificates, and do not fall below our AES-128 threshold.

Features

An issue we have is in deciding which features to implement, or more importantly, which features not to implement. These features often manifest in the range of “extensions”. In some cases there is a difficulty as rarer features can be difficult to test, as they may not arise in the client to Web server setting. Recall that although TLS1.3 is primarily used in the Web server setting (client connects, server downloads HTML), this is not the only setting in which it might be used. Some features in TLS1.3 relate more to these non-HTML applications, which can be negotiated between client and server using the ALPN extension.

One major decision is whether or not to also support the related DTLS 1.3 standards, that is “TLS over datagram transport”. As of now it is recommended not to support DTLS 1.3.

There are some 22 extensions in all. Some of these are deprecated for TLS1.3, some relate only to DTLS1.3, so clearly these will not be implemented at this stage. As of now the client implements about 14 of these. The ones currently not implemented are ones that (a) are not often (or ever) required by Web servers, or (b) are impossible to test until we have our own active TLS 1.3 server. A complicating factor is that many of these extensions can appear in both the clientHello and the serverHello handshake messages, also in the encrypted extensions, handshake request, and as certificate extensions.

Over the coming weeks I intend to implement more of these extensions into the client.