Processing certs and checking sigs

Those following this blog will recall that I am using the information so kindly provided by https://tls13.ulfheim.net/ as a giant Test vector to reverse engineer a path through TLS1.3. As of now I have it working as far down as "Client Application Keys Calc". While there is some wrapping up to do, this is pretty close to the end, as at this stage both parties have determined their shared session key, and the server certificate has been fully processed on the client side.

I had forgotten how complex X.509 certificate processing can be.

The Cert as extracted from a TLS handshake can be as simple as a single self-signed certificate. More likely it is actually a chain of certificates. So at the front there is the host server's certificate, backed by a chain of certs each one signing the next one, ending in a CA self-signed cert. In the example I am working through, its a single certificate, signed with an RSA key belonging to a CA that I don't have access to. In a real world setting, the CA public key would be available through some kind of operating system call.

1. Things are complicated, as from this stage of the protocol everything is encrypted, including the certificate, using a modern AEAD mode of AES (AES-GCM - other choices are possible, but this is the most likely). So first the client has to decrypt the whole server response.
2. Next the client needs to remove the CA signature from the signed cert, and check the validity of the signature by going back along the chain. Since I don't have access to CA public key I had to omit this stage.
3. Next Extract Cert details, like hostname, CA name, county of origin etc
4. Finally extract the host server's public key.

Some of this information (like the host name and URL) gets fed back to the calling application.

I noted that the RSA signature on the certificate in this example uses old fashioned PKCS1.5 padding, a backward compatibility issue, as the modern method for RSA is to use PSS padding. I am not sure if X.509 certs even support PSS padding for new certificates.

As I pointed out last time, none of this information is used to create the actual application session keys. They are now established via Diffie-Hellman methods (and hence can support full forward secrecy). However the whole previous transcript is now hashed and signed by the server using its private key. But since this is new to TLS1.3, this signature, if using RSA, uses modern PSS padding.

This signature now authenticates all that went before, and so the unauthenticate D-H key exchange is now authenticated after-the-fact. Clever!

And so far so good. The client successfully authenticates the TLS_PSS signature using the public key extracted from the certificate, and can now go on to safely use the session key negotiated earlier.

I am also very aware that while I am following a typical example down a common path, there are multiple optional paths that could be followed, and must be supported. For example ECC or RSA. Which curve, which RSA bit length? Which hash function SHA256/384 or 512? So this is going to get quite big.

Its now time to pause the headlong rush, and go back and consolidate the code. However the exercise was well worth it, as I now have a much better understanding of how TLS1.3 works. I will comment the code and start packaging parts of it into a library. One of the design goals is to make the code as compact and fast as possible using only stack memory. That is, Internet of Things Friendly.

Peeking ahead

I am more convinced that we should make the code as in-house as possible (maybe entirely in-house), to retain complete control over it, so that we can develop onwards and outwards to TLS1.4 and TLS2.0. For example most existing implementations use openssl to do their crypto and x.509 cert processing. But openssl is big and bloated, and we want to make this small and tight.