Post Quantum PKI

In a Post Quantum world, assuming that we want to stick with PKI for use with TLS, we will need post-quantum certificate chains, and post-quantum cryptography.

Now the OQS (Open Quantum Safe) project are way ahead of this. They have implemented a patched version of OpenSSL which generates X.509 certificates with public keys and signatures generated using many of the PQ crypto schemes proposed for the NIST competition. For a certificate chain all we will need is a PQ digital signature scheme, so lets choose Dilithium 3. And while we are at it lets choose a KEM which TLS will also need, in this case Kyber768.

Now it may turn out that these are the wrong choices but thanks to our flexible Security Abstraction Layer architecture, and the simple API provided by OQS, it will be extremely easy to switch to some other schemes in the future.

But it gets better. OQS also supply Rust bindings for the PQ crypto, so integration with our Rust code base is also very easy.

A certificate chain

Our Rust server will need a Dilithium based Certificate chain. The chain will have 3 links, the server certificate, the intermediate CA certificate and a root certificate. To get going we will simply add the root certificate to the root certificate store distributed to every client. In the course of the TLS protocol the server will transmit a two link chain, consisting of the server cert and the intermediate cert. To create the components of this chain using the OQS fork of OpenSSL, follow these instructions.

First create the file **`myopenssl.cnf`**

```
[ req ]
distinguished_name       = distinguished_name
extensions               = int_ca
req_extensions           = int_ca
[ int_ca ]
basicConstraints         = CA:TRUE
[ distinguished_name ]
```

Then execute as follows

```
#create Root CA
apps/openssl req -new -x509 -days 365 -newkey dilithium3 -keyout
dilithium3_CA.key -out dilithium3_CA.crt -nodes -subj "/CN=TiigerTLS root CA"
#create Intermediate CA
apps/openssl req -new -newkey dilithium3 -keyout dilithium3_intCA.key -out
dilithium3_intCA.csr -nodes -subj "/CN=TiigerTLS intermediate CA"
apps/openssl x509 -req -days 365 -extfile myopenssl.cnf -extensions int_ca  -in
dilithium3_intCA.csr -CA dilithium3_CA.crt -CAkey dilithium3_CA.key -out
dilithium3_intCA.crt
#create Server certificate
apps/openssl req -new -newkey dilithium3 -keyout dilithium3_server.key -out
dilithium3_server.csr -nodes -subj "/CN=TiigerTLS server"
apps/openssl x509 -req -in dilithium3_server.csr -CA dilithium3_intCA.crt -CAkey
dilithium3_intCA.key -set_serial 01 -days 365 -out dilithium3_server.crt
#verify cert chain
```

```
apps/openssl verify -CAfile dilithium3_CA.crt -untrusted dilithium3_intCA.crt
dilithium3_server.crt
cat dilithium3_server.crt dilithium3_intCA.crt  > dilithium3_certchain.pem
```

Provision the TLS server with files **`dilithium3_server.key`** (private key) and **`dilithium3_certchain.pem`**
Provision the TLS client with **`dilithium3_CA.crt`**, to be inserted into root CA store.

To instead create an elliptic curve certificate chain based on the secp256r1 curve, simply substitute `-newkey dilithium3` everywhere in the above with `-newkey ec:<(openssl ecparam -name prime256v1)`.
Alternatively substitute with `-newkey ec -pkeyopt ec_paramgen_curve:prime256v1` (not tested).

Also you might need the `-CAcreateserial` flag on the third call to openssl.

X.509

The chain consists of X.509 certificates. Fortunately it appears that provisional OIDs have been assigned to the Dilithium scheme (and others), and so our X.509 parsing code had to be patched to recognise the new OID. The work of a few minutes.

And that was more or less all there was to it. The OQS crate had to be added as a dependency to the Rust projects, and lo and behold we have a fully post-quantum TLS1.3 handshake. But the elephant in the room is the size of those elephantine Dilithium3 certificates. Each one is signed and each one embeds a public key. And the public key size is 1184 bytes, and the signature is 3293 bytes. Recall that an elliptic curve public key can be as short as 32 bytes, and a signature 64 bytes.

No-one really has a solution for this. Larger certificates seem to be inevitable, and we will just have to live with them.