Musings on Certificates

I was wondering why some certificates are so big. OK they may use 4096-bit RSA keys and signatures, but that is only 1k bytes. Then there is the text, common names, emails, expiry dates etc. But still hard to see how that could amount to more than another 1k.

Another thing I was thinking about. The whole point of a certificate is to bind a public key with an identity. And that identity should typically be the URL of the server we are connecting to. However the standard name fields (Common Name, Issuer, Subject etc.) are not of this form. Turns out that the URL is in there, in an X.509 extension field called Subject Alternate Names. And its an important part of checking certificate validity to ensure that the hostname of the server can be compared against this, so that we can be sure that this certificate was issued for this website.

First I looked at the www.google.com certificate. And right enough, there in Subject Alternate Names was one entry - www.google.com. So far so sensible. Next I examined a particularly bloated Certificate, when I noticed that the Subject Alternate Names field contained not one URL, but 100! No wonder the certificate was so big! Here is a small section of it, to give a flavour

```
elmothidaelitaliya.com
cdn.haulex.com
beta.manasav.com
monvisogrill.com
www.nibgraphics.com
www.sergenader.com
sipf-innovate.com
sodomind.com
sureshpenikelapati.com
devwebsite.soveluss.com
www.soliddatasolutionsllc.com
www.solomonarnett.com
startupandrew.com
steerthedeer.com
truckinneed.com
```

So what we have here is a single IP address hosting many websites, 100 at a time, and sharing the same certificate (and the same private key). This is web hosting on the cheap, and its not pretty. Good job some-one put a limit of 100 URLs on it, or else certificates would quickly become massive. And its not at all clear to me that one of these websites might not be in a good position to attack one of its neighbours. I certainly would not want to see my e-banking URL on that list.

A quick glance at the Competition

What TLS implementations are already out there? Of course we cannot ignore the elephant in the room – OpenSSL. An amazing heroic multi-year multi-person project which is the daddy and grand-daddy of them all. It is the reference implementation of all things TLS and SSL. It does everything, although it has been a little slow to catch up with AEAD ciphers and the like. Still very actively maintained. It does backwards compatibility all the way back to the year zero. Its been pummelled by cryptanalysts and always bounced back. Like a big old scarred boxer, survivor of a thousand battles, its still standing proud. Its living proof that cryptography can make a difference and can survive sustained attack. Its written in C, and its big. Its open source, and Apache II licensed.

Needless to say we are not attempting to compete with this 500,000 line behemoth. A lot of TLS implementations out there, when you drill down into them, are simply forks of OpenSSL, or make use of OpenSSLs libraries.

At the opposite end of the spectrum, in the very tightly constrained embedded environment there are two main players. BearSSL and SharkSSL. The former is open source, the latter is closed source. As is usual these days in software engineering no sooner has someone tried to make a few honest bucks out of marketing a product, than some schmuck implements the same thing and gives it away for free. And in the crypto world it is specially unfair, since any serious security system integrator is going to want an open source product whose security can be independently verified and audited, and ideally with the most liberal licensing.

Both BearSSL and SharkSSL bend over backwards so that they can shoehorn their code into the smallest possible spaces. The author of BearSSL, Thomas Pornin, a cryptographer of high repute has a great webpage where he openly discusses the evolution of BearSSL. See https://bearssl.org/. This is a great resource. However BearSSL seems to have paused development since 2018, and does not support TLS1.3. It is again written in C.

The commercial competitor SharkSSL is in some ways very similar. Again it seems to have halted development at TLS 1.2. Written in C. The code has found its way into the firmware associated with the ESP8266 SoC, which implements the wifi component of many commercial IoT boards. There is a similar commercial product which also promises a small footprint - Rambus TLS toolkit.

There is a good discussion here - https://lwn.net/Articles/826757/

I would speculate that the reason why these products have not been further developed in recent years and progressed to TLS1.3, is that the space constraints they catered to have considerably eased in the meantime. Larger SoCs like the ESP32 have become standard, and on these devices more complete implementation of TLS become possible. When working in a highly constrained environment there is a temptation to cut back on features, and perhaps use smaller self-signed certificates that do not hold to full PKI security standards.

In response to the loosening of constraints, two other products come into view. Again one is commercial – WolfSSL, and the other is “free” – MbedTLS. Inexplicably MbedTLS despite being strongly supported by ARM does not yet seem to support TLS1.3, despite being promised for years. WolfSSL does support TLS1.3.

So there does appear to be a gap in the market - for an open source TLS1.3 library targetting contemporary embedded devices. And for any decent implementation written in Rust.

Hardware Support

In many cases IoT devices provide hardware support for common cryptographic operations. For example the ATECC608B microchip processor implements nearly all of the crypto (hashing, AES encryption, random number generation, public key operations) required by TLS1.3 (as long as it sticks to elliptic curve cryptography – a common strategy to keep things small, but one that won’t work in a post quantum world). This not only reduces runtime, it also means that a lot less software is required, greatly reducing the footprint for TLS software support.

Unfortunately implementation in hardware means a lack of flexibility. The speed-up is not particularly advantageous for the particular use case that is TLS 1.3. The space savings are probably no longer needed. However hardware support has some advantages. For functions like hashing or

random number generation, which require no secrets, there would be no good reason not to exploit it if it were present. When secrets are involved, hardware enclaves and vaults may have some value, as it should ideally be hard to prise secrets from an IoT board. But that is a bigger discussion for another day.