Building a Security Abstraction Layer (SAL)

Work has been proceeding on integrating other crypto resources, to widen the coverage of our TLS1.3 client, and to allow a choice of crypto providers. As pointed out in our last blog the MIRACL Core library provides enough functionality to permit a fully compliant implementation, but there are some gaps in its coverage of alternative methods, and other libraries may have more mature/faster implementations of some of the cryptographic primitives.

Clearly we would like it to be as simple as possible to unplug a primitive from one provider, and plug in an alternative from another provider. The ideal constraint would be that such a change would have zero impact on any of the other modules that make up the client. In C++ terms we will include in the project build a file **`tls_sal_xxx.cpp`** where **`xxx`** refers to a particular provider, or combination of providers. The associate **`tls_sal.h`** file provides the SAL interface, which is unchanging across all providers. The Security Abstraction Layer file does three main things.

1. Capabilities

There will be functions which report the capabilities of a particular provider combination. For example a function will return a list of symmetric AEAD algorithms that are supported, in the order of the provider's own preference. A provider that used a combination of MIRACL core and the libsodium library might provide a list of three algorithms CHACHA20-POLY1305, AES128-GCM-SHA256, and AES256-GCM-SHA384, in that order, on the basis that the libsodium CHACHA20-POLY1305 implementation is particularly impressive (and MIRACL doesn't support it). In a similar way lists of supported public key signature and key exchange schemes would be provided.

2. Implementations

Functions would be provided to implement all TLS1.3 and X.509 cryptographic algorithms. In many cases these would be empty stubs, as all providers are not required to provide all algorithms (and such empty stubs would never be called as they would not be advertised as capabilities). Nevertheless such algorithms would be provided through a common interface.

3. Data structures

A particular challenge arises in the context of the *transcript hash*, an important component of the TLS 1.3 protocol. Basically its an accumulated hash of the protocol transcript, with the hashing performed by either SHA256 or SHA384, depending on the cipher suite negotiated between the client and the server. The current status of the hash is naturally passed between function calls, as bytes are transmitted and received. One option would be to maintain an appropriate global hash state inside of the Security Abstraction Layer, but for thread safety this would be undesirable.

The solution was to define a universal hash structure, consisting of a blob of memory and an indication of the fixed length of the hash output. That blob of memory would then maintain the hash's current state. Inside of the SAL it would then be caste to the format required for a specific hash function, as specified by the negotiated cipher suite. This effectively solved the problem.

Currently we have defined SALs for a miracl-only version, a miracl+libsodium version, and a miracl+bespoke version. However it should be a simple matter to add alternatives, and hence make our implementation extendable and configurable by the end-user.

Case study

Libsodium supports a nice implementation of the Ed25519 version of the EdDSA digital signature scheme. This is not currently supported by miracl. Therefore we developed a SAL for miracl+libsodium to allow the use of Ed25519 both for TLS1.3 transcript signing, and client-side certificate signature. (This SAL also supports some other relevant libsodium functionality like X25519 key exchange).

In the real world we have not come across any examples of EdDSA being used by server certificates. However it is easy to set up an OpenSSL based local server with a self-signed EdDSA certificate. In the SAL module we added the Ed25519 capability to the list of supported signature algorithms. The client-server protocol then successfully utilised Ed25519 when the protocol was executed.

More to do…

Recall that digital signature is the only public key primitive required outside of the main Diffie-Hellman key exchange. There are 3 main functions required of digital signature in TLS1.3.

(a) Verify the signature on a certificate, given the public key from another certificate. This is required for certificate chain validation. This may require some support for obsoleted algorithms (given that the root CA certificate may be 20 years old).
(b) Verify the Server signature on a hash of the transcript, given the Server's public key extracted from its certificate. This requires support for only a limited number of modern TLS1.3 supported algorithms. Even old algorithms (RSA) can be used in new ways (PSS).
(c) (Optional) Sign a hash of the transcript, given the Client's private key. Again no reason not to use the most up-to-date signature algorithms.

In all cases a range of signature algorithms may be involved. What is needed in the main TLS code are generic SAL interfaces, with actual implementation carried out inside of the SAL.