Roll your own SAL (Security Abstraction Layer)

OK so you don't like the default SALs provided, you like to do your own crypto, or you want to pick-and-mix your own crypto resources from the mass of good quality open source material out there. Or you want to try out some Post Quantum primitives.

Well basically you need to write your own **`tls_sal.cpp`** file, while keeping an eye on the main **`tls1_3.h`** header file and being aware of some of the constants defined there, and maybe changing or adding a few of them. Note that you must **not** change the **`tls_sal.h`** file which defines the immutable API.

You do not have to start this process from scratch. You can use an existing SAL and just change parts of it. We use the MIRACL library for our default SAL, and you can continue to use it as a back-stop to support the bits you are not interested in.

This client implementation is in C++. However as usual C code and headers can be included using the standard construct

```
extern "C" {
     extern int some_c_function(int x, long y);
}
```

Internal functions can be added to your SAL without requiring declaration, as for example

```
static int an_internal_function(int x, long y);
```

A useful resource to have to hand is David Wong's very readable version of the TLS 1.3 standard – see https://www.davidwong.fr/tls13/

Many of the SAL functions make use of an enhanced structure for byte arrays, called an **`octad`**. The reason is that simple arrays in C++ offer no protection against buffer overflow attacks. The **`octad`** attempts to improved on this by providing not only the length of the array, and also a maximum length which will never be exceeded if octads are always properly initialised and processed internally by the approved functions provided in **`tls_octad.h`**

```
typedef struct
{
    int len;   // current length
    int max;   // maximum length
    char *val; // byte array of length max
} octad;
```

SAL Functions you need to provide

1. char *SAL_name()

Return a pointer to a string which describes your SAL

2. bool SAL_initLib()

Perform any global initialization required for your SAL. For example it may be necessary to kick-start a random number generator. Returns *false* if start-up fails.

3. void SAL_endLib()

Do any final tidying up or zeroising of global resources used.

4. int SAL_ciphers(int *ciphers_suites)

Provides a list of the codes for the cipher suites supported by your SAL, and returns the total number supported. See **tls1_3.h** for the possible codes (or you can add new ones). As made clear in the TLS standard, you do not need to support all of them.

5. int SAL_groups(int *groups)

Provides a list of the key exchange groups supported by your SAL, and returns the total number supported. See **tls1_3.h** for the possible codes (or you can add new ones). As made clear in the TLS standard, you do not need to support all of them. Note that the first in the list specifies the group that will be used in the initial TLS client Hello, and you should ensure that any servers you wish to connect with also support this group (otherwise an expensive handshake retry may be needed).

6. int SAL_sigs(int *sigAlgs)

Provides a list of the signature algorithms supported by your SAL, and returns the total number supported. See **tls1_3.h** for the possible codes (or you can add new ones). As made clear in the TLS standard, you do not need to support all of them. Note that these are signatures that a server will apply in the course of the TLS1.3 handshake, and that will be verified using the Server's public key as obtained from a certificate chain supplied by the Server. These algorithms often use modern padding methods.

7. int SAL_sigCerts(int *sigAlgs)

Provides a list of the certificate signature algorithms supported by your SAL, and returns the total number supported. See **tls1_3.h** for the possible codes (or you can add new ones). As made clear in the TLS standard, you do not need to support all of them. Note that these are signatures that will be applied to certificates internal to a certificate chain. These algorithms often use old legacy padding methods (because certificates can be very old). Some signature algorithms may appear in both this and the previous list.

8. int SAL_hashType(int cipher_suite)

Returns an internal code for the hash algorithm used by the given cipher suite.

9. int SAL_hashLen(int hash_type)

Returns the length of the hash algorithm output, for the given hash type.

10. int SAL_aeadKeylen(int cipher_suite)

Returns the length of the AEAD encryption key, for the given cipher suite.

11. int SAL_aeadTaglen(int cipher_suite)

Returns the length of the AEAD authentication tag, for the given cipher suite.

12. int SAL_randomByte()

Returns a single random byte

13. void SAL_randomOctad(int len,octad *R)

Returns a random Octad *R* of the given length *len*, given a pointer to that octad.

14. void SAL_hkdfExtract(int hash_type,octad *PRK,octad *SALT,octad *IKM)

A Key Derivation Function which uses the given hash function to extract a key (*PRK*) from an input salt (*SALT*) and some raw Input Keying Material (*IKM*)

15. void SAL_hkdfExpand(int hash_type, int olen, octad *OKM, octad *PRK, octad *INFO)

A Key Derivation Functions which uses the given hash function to expand a fixed length input key (*PRK*) into a variable length output key (*OKM*) of length olen, given as auxiliary input a text label (*INFO*).

16. void SAL_hmac(int hash_type,octad *T,octad *K,octad *M)

Use the given hash function to calculate an HMAC tag *T*, from a cryptographic key *K* and an input message *M*.

17. void SAL_hashNull(int hash_type,octad *H)

Use the given hash function to calculate the hash *H* of a null string.

18. void SAL_hashInit(int hash_type,unihash *h)

Initialise a hashing context *h* using the given hash function, basically an area of memory used to store the hash functions current state. See **tls1_3.h** for the unihash structure.

19. void SAL_hashProcessArray(unihash *h,char *b,int len)

Process a byte array *b* of length *len* into the given hashing context *h*.

20. int SAL_hashOutput(unihash *h,char *d)

Output final hash from the hash context *h* into a byte array *d*. The length of *d* depends on the hash type, and is returned by this function.

21. void SAL_aeadEncrypt(crypto *send,int hdrlen,char *hdr,int ptlen,char *pt,octad *TAG)

Uses the AEAD algorithm associated with the cryptographic context *send* to AEAD protect a header *hdr* of length *hdrlen* and encrypt in place a plaintext *pt* of length *ptlen*, and to output an authentication tag (*TAG)*. See **tls1_3.h** for the crypto structure. The cryptographic context is set internally from the cipher suite chosen at runtime. Only the cipher suites reported by **SAL_ciphers()** need to be supported.

22. bool SAL_aeadDecrypt(crypto *recv,int hdrlen,char *hdr,int ctlen,char *ct,octad *TAG)

Uses the AEAD algorithm associated with the cryptographic context *recv* to AEAD authenticate a header *hdr* of length *hdrlen*, authenticate and decrypt in place a ciphertext c*t* of length c*tlen*, and to check the input authentication tag *TAG*. If authentication fails the function returns *false*. See **tls1_3.h** for the crypto structure. The cryptographic context is set internally from the cipher suite chosen at runtime. Only the cipher suites reported by **SAL_ciphers()** need to be supported.

23. void SAL_generateKeyPair(int group,octad *SK,octad *PK)

Generate a public/private key pair (*PK/SK*) in the given *group* to support key exchange.

24. void SAL_generateSharedSecret(int group,octad *SK,octad *PK,octad *SS)

Generate a shared secret *SS* from a secret key *SK* and the public key *PK* provided by another party, in the given group.

25. bool SAL_tlsSignatureVerify(int sigAlg,octad *BUFF,octad *SIG,octad *PUBKEY)

Verify the signature *SIG* on a byte array *BUFF* using the public key *PUBKEY*, and using the given signature algorithm.

26. void SAL_tlsSignature(int sigAlg,octad *KEY,octad *BUFF,octad *SIG)

Create a signature *SIG* on the byte array *BUFF* using the private key *KEY*, and using the given signature algorithm.