Node.js `src/crypto` documentation

Welcome. You've found your way to the Node.js native crypto subsystem.

Do not be afraid.

While crypto may be a dark, mysterious, and forboding subject; and while this directory may be filled with many `*.h` and `*.cc` files, finding your way around is not too difficult. And I can promise you that a Gru will not jump out of the shadows and eat you (well, "promise" may be a bit too strong, a Gru may jump out of the shadows and eat you if you live in a place where such things are possible).

Finding your way around

All of the code in this directory is structured into units organized by function or crypto protocol.

The following provide generalized utility declarations that are used throughout the various other crypto files and other parts of Node.js:

- `crypto_util.h` / `crypto_util.cc` (Core crypto definitions)
- `crypto_common.h` / `crypto_common.h` (Shared TLS utility functions)
- `crypto_bio.c` / `crypto_bio.c` (Custom OpenSSL i/o implementation)
- `crypto_groups.h` (modp group definitions)

Of these, `crypto_util.h` and `crypto_util.cc` are the most important, as they provide the core declarations and utility functions used most extensively throughout the rest of the code.

The rest of the files are structured by their function, as detailed in the following table:

File (*.h/*.cc)	Description
`crypto_aes`	AES Cipher support.
`crypto_cipher`	General Encryption/Decryption utilities.
`crypto_clienthello`	TLS/SSL client hello parser implementation. Used during SSL/TLS handshake.
`crypto_context`	Implementation of the `SecureContext` object.
`crypto_dh`	Diffie-Hellman Key Agreement implementation.
`crypto_dsa`	DSA (Digital Signature) Key Generation functions.
`crypto_ec`	Elliptic-curve cryptography implementation.
`crypto_hash`	Basic hash (e.g. SHA-256) functions.
`crypto_hkdf`	HKDF (Key derivation) implementation.
`crypto_hmac`	HMAC implementations.
`crypto_keys`	Utilities for using and generating secret, private, and public keys.
`crypto_pbkdf2`	PBKDF2 key / bit generation implementation.

File (*.h/*.cc)	Description
`crypto_rsa`	RSA Key Generation functions.
`crypto_scrypt`	Scrypt key / bit generation implementation.
`crypto_sig`	General digital signature and verification utilities.
`crypto_spkac`	Netscape SPKAC certificate utilities.
`crypto_ssl`	Implementation of the `SSLWrap` object.
`crypto_timing`	Implementation of the TimingSafeEqual.

When new crypto protocols are added, they will be added into their own `crypto_*.h` and `*.cc` files.

Helpful concepts

Node.js currently uses OpenSSL to provide it's crypto substructure. (Some custom Node.js distributions – such as Electron – use BoringSSL instead.)

This section aims to explain some of the utilities that have been provided to make working with the OpenSSL APIs a bit easier.

Pointer types

Most of the key OpenSSL types need to be explicitly freed when they are no longer needed. Failure to do so introduces memory leaks. To make this easier (and less error prone), the `crypto_util.h` defines a number of smart-pointer aliases that should be used:

```
using X509Pointer = DeleteFnPtr<X509, X509_free>;
using BIOPointer = DeleteFnPtr<BIO, BIO_free_all>;
using SSLCtxPointer = DeleteFnPtr<SSL_CTX, SSL_CTX_free>;
using SSLSessionPointer = DeleteFnPtr<SSL_SESSION, SSL_SESSION_free>;
using SSLPointer = DeleteFnPtr<SSL, SSL_free>;
using PKCS8Pointer = DeleteFnPtr<PKCS8_PRIV_KEY_INFO, PKCS8_PRIV_KEY_INFO_free>;
using EVPKeyPointer = DeleteFnPtr<EVP_PKEY, EVP_PKEY_free>;
using EVPKeyCtxPointer = DeleteFnPtr<EVP_PKEY_CTX, EVP_PKEY_CTX_free>;
using EVPMDPointer = DeleteFnPtr<EVP_MD_CTX, EVP_MD_CTX_free>;
using RSAPointer = DeleteFnPtr<RSA, RSA_free>;
using ECPointer = DeleteFnPtr<EC_KEY, EC_KEY_free>;
using BignumPointer = DeleteFnPtr<BIGNUM, BN_free>;
using NetscapeSPKIPointer = DeleteFnPtr<NETSCAPE_SPKI, NETSCAPE_SPKI_free>;
using ECGroupPointer = DeleteFnPtr<EC_GROUP, EC_GROUP_free>;
using ECPointPointer = DeleteFnPtr<EC_POINT, EC_POINT_free>;
using ECKeyPointer = DeleteFnPtr<EC_KEY, EC_KEY_free>;
using DHPointer = DeleteFnPtr<DH, DH_free>;
using ECDSASigPointer = DeleteFnPtr<ECDSA_SIG, ECDSA_SIG_free>;
using HMACCtxPointer = DeleteFnPtr<HMAC_CTX, HMAC_CTX_free>;
using CipherCtxPointer = DeleteFnPtr<EVP_CIPHER_CTX, EVP_CIPHER_CTX_free>;
```

Examples of these being used are pervasive through the `src/crypto` code.

`ByteSource`

The `ByteSource` class is a helper utility representing a *read-only* byte array. Instances can either wrap external ("foreign") data sources, such as an `ArrayBuffer` (`v8::BackingStore`) or allocated data. If allocated data is used, then the allocation is freed automatically when the `ByteSource` is destroyed.

`ArrayBufferOrViewContents`

The `ArrayBufferOfViewContents` class is a helper utility that abstracts `ArrayBuffer`, `TypedArray`, or `DataView` inputs and provides access to their underlying data pointers. It is used extensively through `src/crypto` to make it easier to deal with inputs that allow any `ArrayBuffer`-backed object.

`AllocatedBuffer`

The `AllocatedBuffer` utility is defined in `allocated_buffer.h` and is not specific to `src/crypto`. It is used extensively within `src/crypto` to hold allocated data that is intended to be output in response to various crypto functions (generated hash values, or ciphertext, for instance).

Currently, we are working to transition away from using `AllocatedBuffer` to directly using the `v8::BackingStore` API. This will take some time. New uses of `AllocatedBuffer` should be avoided if possible.

Key objects

Most crypto operations involve the use of keys – cryptographic inputs that protect data. There are three general types of keys:

- Secret Keys (Symmetric)
- Public Keys (Asymmetric)
- Private Keys (Asymmetric)

Secret keys consist of a variable number of bytes. They are "symmetrical" in that the same key used to encrypt data, or generate a signature, must be used to decrypt or validate that signature. If two people are exchanging messages encrypted using a secret key, both of them must have access to the same secret key data.

Public and Private keys always come in pairs. When one is used to encrypt data or generate a signature, the other is used to decrypt or validate the signature. The Public key is intended to be shared and can be shared openly. The Private key must be kept secret and known only to the owner of the key.

The `src/crypto` subsystem uses several objects to represent keys. These objects are structured in a way to allow key data to be shared across multiple threads

(the Node.js main thread, Worker Threads, and the libuv threadpool).

Refer to `crypto_keys.h` and `crypto_keys.cc` for all code relating to the core key objects.

`ManagedEVPPKey` The `ManagedEVPPKey` class is a smart pointer for OpenSSL `EVP_PKEY` structures. These manage the lifecycle of Public and Private key pairs.

`KeyObjectData` `KeyObjectData` is an internal thread-safe structure used to wrap either a `ManagedEVPPKey` (for Public or Private keys) or a `ByteSource` containing a Secret key.

`KeyObjectHandle` The `KeyObjectHandle` provides the interface between the native C++ code handling keys and the public JavaScript `KeyObject` API.

`KeyObject` A `KeyObject` is the public Node.js-specific API for keys. A single `KeyObject` wraps exactly one `KeyObjectHandle`.

`CryptoKey` A `CryptoKey` is the Web Crypto API's alternative to `KeyObject`. In the Node.js implementation, `CryptoKey` is a thin wrapper around the `KeyObject` and it is largely possible to use them interchangeably.

`CryptoJob`

All operations that are not either Stream-based or single-use functions are built around the `CryptoJob` class.

A `CryptoJob` encapsulates a single crypto operation that can be invoked synchronously or asynchronously.

The `CryptoJob` class itself is a C++ template that takes a single `CryptoJobTraits` struct as a parameter. The `CryptoJobTraits` provides the implementation detail of the job.

There are (currently) four basic `CryptoJob` specializations:

- `CipherJob` (defined in `src/crypto_cipher.h`) – Used for encrypt and decrypt operations.
- `KeyGenJob` (defined in `src/crypto_keygen.h`) – Used for secret and key pair generation operations.
- `KeyExportJob` (defined in `src/crypto_keys.h`) – Used for key export operations.
- `DeriveBitsJob` (defined in `src/crypto_util.h`) – Used for key and byte derivation operations.

Every `CryptoJobTraits` provides two fundamental operations:

- Configuration – Processes input arguments when a `CryptoJob` instance is created.

- Implementation – Provides the specific implementation of the operation.

The Configuration is typically provided by an `AdditionalConfig()` method, the signature of which is slightly different for each of the above `CryptoJob` specializations. Despite the signature differences, the purpose of the `AdditionalConfig()` function remains the same: to process input arguments and set the properties on the `CryptoJob`'s parameters object.

The parameters object is specific to each `CryptoJob` type, and is stored with the `CryptoJob`. It holds all of the inputs that are used by the Implementation. The inputs held by the parameters must be threadsafe.

The `AdditionalConfig()` function is always called when the `CryptoJob` instance is being created.

The Implementation function is unique to each of the `CryptoJob` specializations and will either be called synchronously within the current thread or from within the libuv threadpool.

Every `CryptoJob` instance exposes a `run()` function to the JavaScript layer. When called, `run()` with either dispatch the job to the libuv threadpool or invoke the Implementation function synchronously. If invoked synchronously, run() will return a JavaScript array. The first value in the array is either an `Error` or `undefined`. If the operation was successful, the second value in the array will contain the result of the operation. Typically, the result is an `ArrayBuffer`, but certain `CryptoJob` types can alter the output.

If the `CryptoJob` is processed asynchronously, then the job must have an `ondone` property whose value is a function that is invoked when the operation is complete. This function will be called with two arguments. The first is either an `Error` or `undefined`, and the second is the result of the operation if successful.

For `CipherJob` types, the output is always an `ArrayBuffer`.

For `KeyExportJob` types, the output is either an `ArrayBuffer` or a JavaScript object (for JWK output format);

For `KeyGenJob` types, the output is either a single KeyObject, or an array containing a Public/Private key pair represented either as a `KeyObjectHandle` object or a `Buffer`.

For `DeriveBitsJob` type output is typically an `ArrayBuffer` but can be other values (`RandomBytesJob` for instance, fills an input buffer and always returns `undefined`).

Errors

`ThrowCryptoError` and the `THROW_ERR_CRYPTO_*` macros The `ThrowCryptoError()` is a legacy utility that will throw a JavaScript exception containing details collected from OpenSSL about a failed operation. `ThrowCryptoError()` should only be used when necessary to report low-level OpenSSL failures.

In `node_errors.h`, there are a number of `ERR_CRYPTO_*` macro definitions that define semantically specific errors. These can be called from within the C++ code as functions, like `THROW_ERR_CRYPTO_INVALID_IV(env)`. These methods should be used to throw JavaScript errors when necessary.

Crypto API patterns

Operation mode

All crypto functions in Node.js operate in one of three modes:

- Synchronous single-call
- Asynchronous single-call
- Stream-oriented

It is often possible to perform various operations across multiple modes. For instance, cipher and decipher operations can be performed in any of the three modes.

Synchronous single-call operations are always blocking. They perform their actions immediately.

```
// Example synchronous single-call operation
const a = new Uint8Array(10);
const b = new Uint8Array(10);
crypto.timingSafeEqual(a, b);
```

Asynchronous single-call operations generally perform a number of synchronous input validation steps, but then defer the actual crypto-operation work to the libuv threadpool.

```
// Example asynchronous single-call operation
const buf = new Uint8Array(10);
crypto.randomFill(buf, (err, buf) => {
  console.log(buf);
});
```

For the legacy Node.js crypto API, asynchronous single-call operations use the traditional Node.js callback pattern, as illustrated in the previous `randomFill()` example. In the Web Crypto API (accessible via `require('crypto').webcrypto`), all asynchronous single-call operations are Promise-based.

```
// Example Web Crypto API asynchronous single-call operation
const { subtle } = require('crypto').webcrypto;

subtle.generateKeys({ name: 'HMAC', length: 256 }, true, ['sign'])
  .then((key) => {
    console.log(key);
  })
  .catch((error) => {
```

```
    console.error('an error occurred');
  });
```

In nearly every case, asynchronous single-call operations make use of the libuv threadpool to perform crypto operations off the main event loop thread.

Stream-oriented operations use an object to maintain state over multiple individual synchronous steps. The steps themselves can be performed over time.

```
// Example stream-oriented operation
const hash = crypto.createHash('sha256');
let updates = 10;
setTimeout(() => {
  hash.update('hello world');
  setTimeout(() => {
    console.log(hash.digest();)
  }, 1000);
}, 1000);
```