Notes on engines of 2001-09-24

This "description" (if one chooses to call it that) needed some major updating so here goes. This update addresses a change being made at the same time to OpenSSL, and it pretty much completely restructures the underlying mechanics of the "ENGINE" code. So it serves a double purpose of being a "ENGINE internals for masochists" document *and* a rather extensive commit log message. (I'd get lynched for sticking all this in CHANGES.md or the commit mails :-).

ENGINE_TABLE underlies this restructuring, as described in the internal header "eng_local.h", implemented in eng_table.c, and used in each of the "class" files; tb_rsa.c, tb_dsa.c, etc.

However, "EVP_CIPHER" underlies the motivation and design of ENGINE_TABLE so I'll mention a bit about that first. EVP_CIPHER (and most of this applies equally to EVP_MD for digests) is both a "method" and a algorithm/mode identifier that, in the current API, "lingers". These cipher description + implementation structures can be defined or obtained directly by applications, or can be loaded "en masse" into EVP storage so that they can be catalogued and searched in various ways, ie. two ways of encrypting with the "des_cbc" algorithm/mode pair are;

```
(i) directly;
     const EVP_CIPHER *cipher = EVP_des_cbc();
     EVP_EncryptInit(&ctx, cipher, key, iv);
     [ ... use EVP_EncryptUpdate() and EVP_EncryptFinal() ...]

(ii) indirectly;
     OpenSSL_add_all_ciphers();
     cipher = EVP_get_cipherbyname("des_cbc");
     EVP_EncryptInit(&ctx, cipher, key, iv);
     [ ... etc ... ]
```

The latter is more generally used because it also allows ciphers/digests to be looked up based on other identifiers which can be useful for automatic cipher selection, eg. in SSL/TLS, or by user-controllable configuration.

The important point about this is that EVP_CIPHER definitions and structures are passed around with impunity and there is no safe way, without requiring massive rewrites of many applications, to assume that EVP_CIPHERs can be reference counted. One an EVP_CIPHER is exposed to the caller, neither it nor anything it comes from can "safely" be destroyed. Unless of course the way of getting to such ciphers is via entirely distinct API calls that didn't exist before. However existing API usage cannot be made to understand when an EVP_CIPHER pointer, that has been passed to the caller, is no longer being used.

The other problem with the existing API w.r.t. to hooking EVP_CIPHER support into ENGINE is storage - the OBJ_NAME-based storage used by EVP to

register ciphers simultaneously registers cipher *types* and cipher *implementations* - they are effectively the same thing, an "EVP_CIPHER" pointer. The problem with hooking in ENGINEs is that multiple ENGINEs may implement the same ciphers. The solution is necessarily that ENGINE-provided ciphers simply are not registered, stored, or exposed to the caller in the same manner as existing ciphers. This is especially necessary considering the fact ENGINE uses reference counts to allow for cleanup, modularity, and DSO support - yet EVP_CIPHERs, as exposed to callers in the current API, support no such controls.

Another sticking point for integrating cipher support into ENGINE is linkage. Already there is a problem with the way ENGINE supports RSA, DSA, etc whereby they are available *because* they're part of a giant ENGINE called "openssl". Ie. all implementations *have* to come from an ENGINE, but we get round that by having a giant ENGINE with all the software support encapsulated. This creates linker hassles if nothing else - linking a 1-line application that calls 2 basic RSA functions (eg. "RSA_free(RSA_new());") will result in large quantities of ENGINE code being linked in *and* because of that DSA, DH, and RAND also. If we continue with this approach for EVP_CIPHER support (even if it *was* possible) we would lose our ability to link selectively by selectively loading certain implementations of certain functionality. Touching any part of any kind of crypto would result in massive static linkage of everything else. So the solution is to change the way ENGINE feeds existing "classes", ie. how the hooking to ENGINE works from RSA, DSA, DH, RAND, as well as adding new hooking for EVP_CIPHER, and EVP_MD.

The way this is now being done is by mostly reverting back to how things used to work prior to ENGINE :-). Ie. RSA now has a "RSA_METHOD" pointer again - this was previously replaced by an "ENGINE" pointer and all RSA code that required the RSA_METHOD would call ENGINE_get_RSA() each time on its ENGINE handle to temporarily get and use the ENGINE's RSA implementation. Apart from being more efficient, switching back to each RSA having an RSA_METHOD pointer also allows us to conceivably operate with *no* ENGINE. As we'll see, this removes any need for a fallback ENGINE that encapsulates default implementations - we can simply have our RSA structure pointing its RSA_METHOD pointer to the software implementation and have its ENGINE pointer set to NULL.

A look at the EVP_CIPHER hooking is most explanatory, the RSA, DSA (etc) cases turn out to be degenerate forms of the same thing. The EVP storage of ciphers, and the existing EVP API functions that return "software" implementations and descriptions remain untouched. However, the storage takes more meaning in terms of "cipher description" and less meaning in terms of "implementation". When an EVP_CIPHER_CTX is actually initialised with an EVP_CIPHER method and is about to begin en/decryption, the hooking to ENGINE comes into play. What happens is that cipher-specific ENGINE code is asked for an ENGINE pointer (a functional reference) for any ENGINE that is registered to perform the algo/mode that the provided EVP_CIPHER structure

represents. Under normal circumstances, that ENGINE code will return NULL because no ENGINEs will have had any cipher implementations *registered*. As such, a NULL ENGINE pointer is stored in the EVP_CIPHER_CTX context, and the EVP_CIPHER structure is left hooked into the context and so is used as the implementation. Pretty much how things work now except we'd have a redundant ENGINE pointer set to NULL and doing nothing.

Conversely, if an ENGINE *has* been registered to perform the algorithm/mode combination represented by the provided EVP_CIPHER, then a functional reference to that ENGINE will be returned to the EVP_CIPHER_CTX during initialisation. That functional reference will be stored in the context (and released on cleanup) - and having that reference provides a *safe* way to use an EVP_CIPHER definition that is private to the ENGINE. Ie. the EVP_CIPHER provided by the application will actually be replaced by an EVP_CIPHER from the registered ENGINE - it will support the same algorithm/mode as the original but will be a completely different implementation. Because this EVP_CIPHER isn't stored in the EVP storage, nor is it returned to applications from traditional API functions, there is no associated problem with it not having reference counts. And of course, when one of these "private" cipher implementations is hooked into EVP_CIPHER_CTX, it is done whilst the EVP_CIPHER_CTX holds a functional reference to the ENGINE that owns it, thus the use of the ENGINE's EVP_CIPHER is safe.

The "cipher-specific ENGINE code" I mentioned is implemented in tb_cipher.c but in essence it is simply an instantiation of "ENGINE_TABLE" code for use by EVP_CIPHER code. tb_digest.c is virtually identical but, of course, it is for use by EVP_MD code. Ditto for tb_rsa.c, tb_dsa.c, etc. These instantiations of ENGINE_TABLE essentially provide linker-separation of the classes so that even if ENGINEs implement *all* possible algorithms, an application using only EVP_CIPHER code will link at most code relating to EVP_CIPHER, tb_cipher.c, core ENGINE code that is independent of class, and of course the ENGINE implementation that the application loaded. It will *not* however link any class-specific ENGINE code for digests, RSA, etc nor will it bleed over into other APIs, such as the RSA/DSA/etc library code.

ENGINE_TABLE is a little more complicated than may seem necessary but this is mostly to avoid a lot of "init()"-thrashing on ENGINEs (that may have to load DSOs, and other expensive setup that shouldn't be thrashed unnecessarily) *and* to duplicate "default" behaviour. Basically an ENGINE_TABLE instantiation, for example tb_cipher.c, implements a hash-table keyed by integer "nid" values. These nids provide the uniquenness of an algorithm/mode - and each nid will hash to a potentially NULL "ENGINE_PILE". An ENGINE_PILE is essentially a list of pointers to ENGINEs that implement that particular 'nid'. Each "pile" uses some caching tricks such that requests on that 'nid' will be cached and all future requests will return immediately (well, at least with minimal operation) unless a change is made to the pile, eg. perhaps an ENGINE was unloaded. The reason is that an application could have support for 10 ENGINEs statically

linked in, and the machine in question may not have any of the hardware those 10 ENGINEs support. If each of those ENGINEs has a "des_cbc" implementation, we want to avoid every EVP_CIPHER_CTX setup from trying (and failing) to initialise each of those 10 ENGINEs. Instead, the first such request will try to do that and will either return (and cache) a NULL ENGINE pointer or will return a functional reference to the first that successfully initialised. In the latter case it will also cache an extra functional reference to the ENGINE as a "default" for that 'nid'. The caching is acknowledged by a 'uptodate' variable that is unset only if un/registration takes place on that pile. Ie. if implementations of "des_cbc" are added or removed. This behaviour can be tweaked; the ENGINE_TABLE_FLAG_NOINIT value can be passed to ENGINE_set_table_flags(), in which case the only ENGINEs that tb_cipher.c will try to initialise from the "pile" will be those that are already initialised (ie. it's simply an increment of the functional reference count, and no real "initialisation" will take place).

RSA, DSA, DH, and RAND all have their own ENGINE_TABLE code as well, and the difference is that they all use an implicit 'nid' of 1. Whereas EVP_CIPHERs are actually qualitatively different depending on 'nid' (the "des_cbc" EVP_CIPHER is not an interoperable implementation of "aes_256_cbc"), RSA_METHODs are necessarily interoperable and don't have different flavours, only different implementations. In other words, the ENGINE_TABLE for RSA will either be empty, or will have a single ENGINE_PILE hashed to by the 'nid' 1 and that pile represents ENGINEs that implement the single "type" of RSA there is.

Cleanup - the registration and unregistration may pose questions about how cleanup works with the ENGINE_PILE doing all this caching nonsense (ie. when the application or EVP_CIPHER code releases its last reference to an ENGINE, the ENGINE_PILE code may still have references and thus those ENGINEs will stay hooked in forever). The way this is handled is via "unregistration". With these new ENGINE changes, an abstract ENGINE can be loaded and initialised, but that is an algorithm-agnostic process. Even if initialised, it will not have registered any of its implementations (to do so would link all class "table" code despite the fact the application may use only ciphers, for example). This is deliberately a distinct step. Moreover, registration and unregistration has nothing to do with whether an ENGINE is *functional* or not (ie. you can even register an ENGINE and its implementations without it being operational, you may not even have the drivers to make it operate). What actually happens with respect to cleanup is managed inside eng_lib.c with the `engine_cleanup_***` functions. These functions are internal-only and each part of ENGINE code that could require cleanup will, upon performing its first allocation, register a callback with the "engine_cleanup" code. The other part of this that makes it tick is that the ENGINE_TABLE instantiations (tb_***.c) use NULL as their initialised state. So if RSA code asks for an ENGINE and no ENGINE has registered an implementation, the code will simply return NULL and the tb_rsa.c state will be unchanged. Thus, no cleanup is required unless registration

takes place. ENGINE_cleanup() will simply iterate across a list of registered cleanup callbacks calling each in turn, and will then internally delete its own storage (a STACK). When a cleanup callback is next registered (eg. if the cleanup() is part of a graceful restart and the application wants to cleanup all state then start again), the internal STACK storage will be freshly allocated. This is much the same as the situation in the ENGINE_TABLE instantiations ... NULL is the initialised state, so only modification operations (not queries) will cause that code to have to register a cleanup.

What else? The bignum callbacks and associated ENGINE functions have been removed for two obvious reasons; (i) there was no way to generalise them to the mechanism now used by RSA/DSA/..., because there's no such thing as a BIGNUM method, and (ii) because of (i), there was no meaningful way for library or application code to automatically hook and use ENGINE supplied bignum functions anyway. Also, ENGINE_cpy() has been removed (although an internal-only version exists) - the idea of providing an ENGINE_cpy() function probably wasn't a good one and now certainly doesn't make sense in any generalised way. Some of the RSA, DSA, DH, and RAND functions that were fiddled during the original ENGINE changes have now, as a consequence, been reverted back. This is because the hooking of ENGINE is now automatic (and passive, it can internally use a NULL ENGINE pointer to simply ignore ENGINE from then on).

Hell, that should be enough for now ... comments welcome.