# OPENSSL cheat sheet

Cédric Lauradoux
cedric.lauradoux@inria.fr

## Overview

The `openssl` utility has 46 commands which can be used to perform many cryptographic operations. The commands can be classify into 7 categories:

| Version | | | | |
|---|---|---|---|---|
| `version` | `ciphers` | | `engine` | `errstr` |
| Benchmarking | | | | |
| `speed` | `s_time` | | | |
| Symmetric encryption and hashing | | | | |
| `enc` | `rand` | `dgst` | `passwd` | |
| Asymmetric encryption and signature | | | | |
| `gendh` | `gendsa` | `genpkey` | `genrsa` | `pkey` |
| `dh` | `dsa` | `rsa` | `ec` | `prime` |
| `dhparam` | `dsaparam` | `rsautl` | `ecparam` | `pkeyutl` |
| `pkeyparam` | | | | |
| PKI and certificat | | | | |
| `ca` | `crl` | `crl2pkcs7` | `nseq` | `ocsp` |
| `pkcs12` | `pkcs7` | `pkcs8` | `req` | `spkac` |
| `x509` | | | `verify` | |
| Server and session | | | | |
| `asn1parse` | `s_client` | `s_server` | `srp` | `ts` |
| `sess_id` | | | | |
| S/Mine | | | | |
| `smime` | `cms` | | | |

Most of the commands are associated to asymmetric cryptography (16 commands) and PKI (12 commands). All the commands are used with options and flags. For instance, to determine the version of `openssl` you are using:

```
$ openssl version
```

If you want to have all the details on the `openssl` you are using, you just need to add the following flag:

```
$ openssl version -a
```

The following command gives all the cipher suites available in `openssl`:

```
$ openssl ciphers -v
```

To know if `openssl` has been setup with special cryptographic engines (hardware modules or CPU instruction set extension) use:

```
$ openssl engine
```

If you need help to understand a command, you can find help in the `man` or you can use the command with `-help`.

```
$ man enc
$ openssl enc -help
```

Actually, there is no `-help` flag in `openssl` but this is an invalid command that will display all the options and flags for the command.

## Symmetric Encryption and hashing

### Random number generation

The `rand` command is very useful to produce symmetric keys, initialization vectors or nonces. It produces pseudo-random bytes with

- `num` number of bytes to produce
- `-out file` write the byte in the file `file` *[optional]*
- `-rand seed` use a seed stored in `seed` *[optional]*
- `-base64` output written in base64 *[optional]*
- `-hex` output written in hexadecimal *[optional]*

The following command produces a 256-bit random value written in hexadecimal stored in the file `mykey.key`:

```
$ openssl rand 32 -out mykey.key -hex
```

### Encryption

The `enc` command is used to encrypt file using symmetric ciphers. `enc` has 2 main options:

- `-e` for encryption or `-d` for decryption,
- `-ciphername` the cipher choice with key length, mode of operation and possible authentication mode.

The beginning of the command to encrypt a file with the AES 128-bit in CFB mode is:

```
$ openssl enc -e -aes-128-cfb
```

Note that it is now even possible to use authenticated mode like CCM or GCM.
By default the input and output of the `enc` command are the standard input and the standard output of the terminal. It can be changed using the following options:

- `-in input` with `input` is the filename to process,
- `-out output` the result is stored in `output`.

```
$ openssl enc -e -in file.in -out file.out
```

The `enc` command has 2 modes:

- classical mode with key,
- password mode with a password used to encrypt the file.

The classical mode of the `enc` command need two extra parameters:

- `-K key` with `key` the hexadecimal value of the key,
- `-iv iv` with `iv` the hexadecimal value of the initialization vector.

A full command for the classical mode looks like:

```
$ openssl enc -e -aes-128-cfb -in input.clear
-out output.enc -K key -iv iv
```

The password mode has 4 options:

- `-k passwrd` with `passwrd` an `ascii` password,
- `-S salt` with `salt` the hexadecimal value of a salt.
- `-nosalt`
- `-p` print the key and iv generated from the password

Without the option `-k`, the prompt asks the user for a password (and confirmation). The most basic command for the password mode is:

```
$ openssl enc -e -aes-128-cfb -in input.clear
-out output.enc
```

It is important to notice that even when it is not needed, the `enc` command always add some extra padding to the plaintext. To get rid of this padding, you can use `-nopad` option.

### Hashing and Authentication

The command `dgst` can be used to compute digests of files and authentication tags. The main option of this command is the choice of the message digest algorithm:

```
$ openssl dgst -alg myfile
```

is the command to compute the digest of `myfile` using algorithm `alg`. For SHA256, we have for instance:

```
$ openssl dgst -sha256 myfile
```

There is 4 options to format the output of `dgst`:

- `-hex` for a digest in hexadecimal *[optional]*
- `-binary` for a digest in binary *[optional]*
- `-r` for an output similar to `md5sum` command *[optional]*
- `-out file` to output in `file` *[optional]*

The `dgst` command proposes also an option to use HMAC (there is also an option `-mac` to use any type of MAC algorithms but no algorithm are available by default):

```
$ openssl dgst -sha256 -hmac KEY myfile
```

with `KEY` the hexadecimal value of the key.

### Password fingerprint

The `passwd` command produces password digests for authentication schemes using Unix format (`-crypt`), MD5 (`-1`) or Apache (`apr1`). The following command produces the digests of each line of file `passfile`:

```
$ openssl passwd -crypt -salt AA -in passfile
```

The option `-salt XX` is used to create all the digests using a ASCII character salt `XX`. If this option is not used the salt is chosen randomly for each line of the file.

## Benchmarking

The `speed` command can be used to measure the speed of different ciphers. The list of available ciphers for benchmark is short compared to the list of all the cipher suites available in `openssl`. Use `speed -help` to get this list. The following command benchmark all the ciphers:

```
$ openssl speed
```

This command only benchmarks AES modes of operation:

```
$ openssl speed aes
```

## Asymmetric Encryption

We use the example of RSA but the same approach works for DSA and elliptic curves.

### Key generation

To produce an RSA keys-pair, the command is `genrsa`. This main argument of this command is the key size given in bits.

```
openssl genrsa 2048
```

The main options of this command are related to the format of the output:

- `-out file` output the key in `file`
- `-des`, `-aes128`, `-aes192`,..., `-aes256` encrypt the key file with a symmetric block cipher in CBC mode with a password. *[optional]*

The command

```
openssl genrsa 2048 -out mykey.pem -aes128
```

will produce a 2048-bit public and private key and store it into `mykey.pem` which is encrypted with AES-128 in CBC mode.

To extract the public key from `mykey.pem`, we use the command `rsa`. The format of the input and output of this command can be specified by the arguments:

- `-in file.in` the key pairs is located in `file.in`
- `-inform DER|NET|PEM` input format *[optional]*
- `-out file.out` the output is stored in `file.out`
- `-outform DER|NET|PEM` output format *[optional]*
- `-pubout` to extract the public key only *[optional]*

```
openssl rsa -in mykey.pem -outform PEM\
-pubout -out public.pem
```

### Encryption/decryption

To encrypt a file using our newly generated RSA keys, we use the `rsautl` command with `-encrypt` and `-decrypt` option. The following command use the standard input and output with instead of the usual `-in` and `-out` option available in most `openssl` commands.

```
echo 'Mr Lauradoux is a genius!' |\
openssl rsautl -encrypt -pubin -inkey\
public.pem > file.out
```

It is important to notice the usage of `-pubin` to tell `rsautl` that the input key file contains only a public key. Without this option, `rsautl` expects a input key file with a private and a public key.
Padding is necessary with RSA. By default it uses PKCS#1 v1.5 but you can use any of these 4 padding schemes: `-pkcs`, `-oaep`, `-ssl`, `-raw`.
To decrypt with the private key:

```
openssl rsautl -decrypt -inkey mykey.pem\
 -in file.out
```

### Signatures

To sign and verify a signature, we still use `rsautl` but this time with the `-sign` and `-verify` option.
We sign with the private key:

```
echo 'Mr Lauradoux is stronger than Chuck\
 Norris!' | openssl rsautl -sign  -inkey\
 mykey.pem -out file.out
```

and verify with the

```
openssl rsautl -verify  -pubin -inkey\
 public.pem -in file.out
```

### Limitations

It is not possible to encrypt or sign large files with asymmetric cryptosystems. To circumvent this problem, we use **hybrid encryption** and hashing for signature.
To sign a large file, we first compute its cryptographic digest and then sign the digest. Because the output of the `dgst` is not convenient for chaining `rsautl`, two options have been added to `dgst`:

- `-sign private.pem` to sign a digest with the private key in `private.pem` *[optional]*
- `-verify public.pem` to verify the signature of a digest with the public key `public.pem` *[optional]*

Hybrid encryption is not that easy to perform. You first need to produce a key `key` and an initialization vector `iv` with `rand`.
Encrypt the large file using `enc` using `key` and `iv`.
Finally, you can encrypt using the user public key `public.pem` the value `key`.
We can then send the `iv` (public), the encrypted large file and the encrypted version of `key`. The receiver can use its private key to recover `key` and use it with `iv` to recover the large file.