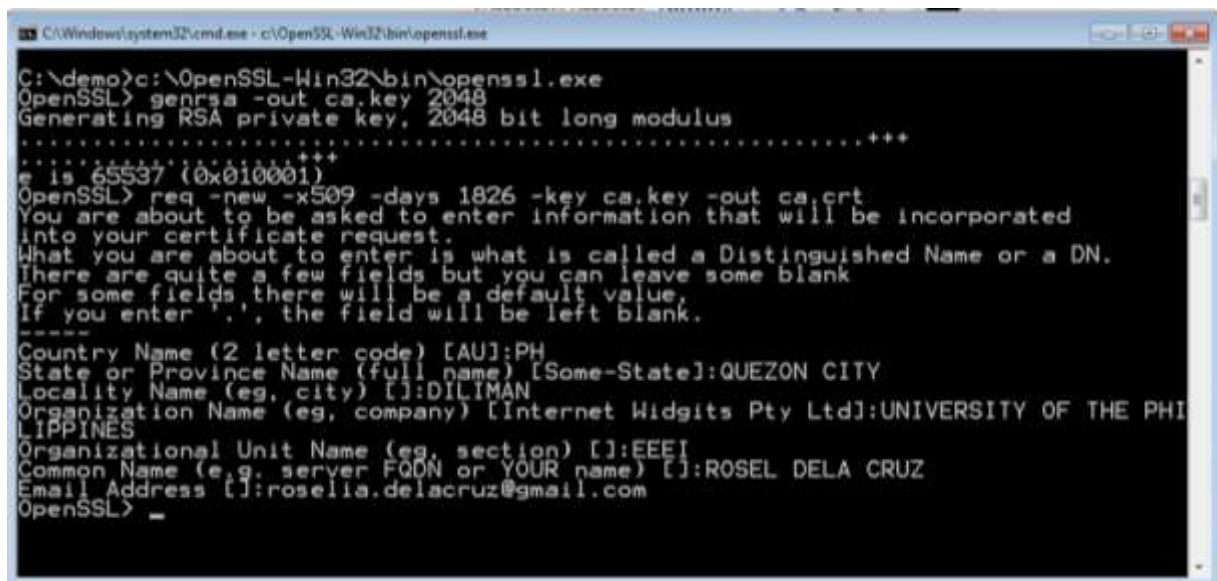


CS 253 COMPUTER SECURITY ENCRYPTION/DECRYPTION USING OPENSSL

Roselia G. Dela Cruz
2010-81736

Pre-requisites:

1. Download OpenSSL from <https://www.openssl.org> for Windows
2. Installed Win32 OpenSSL v1.1.0e. Run OpenSSL.
3. C++ Compiler



```
C:\demo>c:\OpenSSL-Win32\bin\openssl.exe
OpenSSL> genrsa -out ca.key 2048
Generating RSA private key, 2048 bit long modulus
.....+++
e is 65537 (0x010001)
OpenSSL> req -new -x509 -days 1826 -key ca.key -out ca.crt
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:PH
State or Province Name (full name) [Some-State]:QUEZON CITY
Locality Name (eg, city) []:DILIMAN
Organization Name (eg, company) [Internet Widgits Pty Ltd]:UNIVERSITY OF THE PHI
LIPPINES
Organizational Unit Name (eg, section) []:EEEI
Common Name (e.g. server FQDN or YOUR name) []:ROSEL DELA CRUZ
Email Address []:roselia.delacruz@gmail.com
OpenSSL> _
```

A. SYMMETRIC ENCRYPTION.

Symmetric AES encryption on the 512x512 Color (24-bit) Lena image
(<http://www.ece.rice.edu/~wakin/images/lena512color.tiff>)

- ✓ Next step: create our subordinate CA that will be used for the actual signing. First, generate the key:

```
OpenSSL> genrsa -out ia.key 2048
```



```
C:\Windows\system32\cmd.exe - c:\OpenSSL-Win32\bin\openssl.exe
OpenSSL> genrsa -out ia.key 2048
Generating RSA private key, 2048 bit long modulus
.....+++
e is 65537 (0x010001)
```

- ✓ Get the public key. Let the other party send you a certificate or their public key. If they send to a certificate you can extract the public key using this command:

```
openssl rsa -in cakey.pem -out publickey.pem -outform PEM -pubout
```

- ✓ Generate the random password file. Use the following command to generate the random key:

```
openssl rand -base64 -out key.bin 128
```



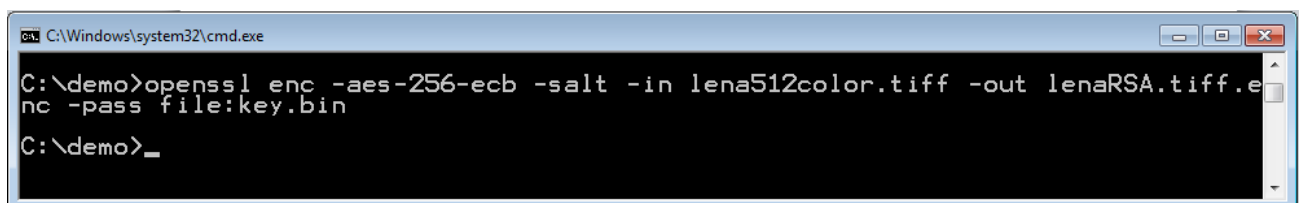
```
C:\Windows\system32\cmd.exe
C:\demo>openssl rand -help
Usage: rand [flags] num
Valid options are:
  -help          Display this summary
  -out outfile   Output file
  -rand val      Load the file(s) into the random number generator
  -base64        Base64 encode output
  -hex           Hex encode output
  -engine val    Use engine, possibly a hardware device
C:\demo>openssl rand -base64 -out key.bin 128
C:\demo>
```

- ✓ Do this every time you encrypt a file. Use a new key every time!

A.1 Using ECB mode, AES 128

- ✓ Encrypt the file with the random key. Use the following command to encrypt the image (Lena) file with the random key:

```
openssl enc -aes-256-cbc -salt -in lena512color.tiff -out lenaRSA.tiff.enc -pass file:key.bin
```

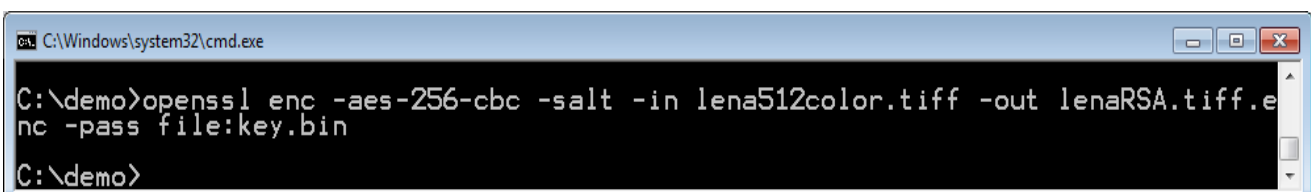


```
C:\Windows\system32\cmd.exe
C:\demo>openssl enc -aes-256-ecb -salt -in lena512color.tiff -out lenaRSA.tiff.e
nc -pass file:key.bin
C:\demo>_
```

A.2 Using CBC mode, AES-128

- ✓ Encrypt the file with the random key. Use the following command to encrypt the image (Lena) file with the random key:

```
openssl enc -aes-256-cbc -salt -in lena512color.tiff -out lenaRSA.tiff.enc -pass file:key.bin
```



```
C:\Windows\system32\cmd.exe
C:\demo>openssl enc -aes-256-cbc -salt -in lena512color.tiff -out lenaRSA.tiff.e
nc -pass file:key.bin
C:\demo>
```

B. HASHING

Hashing Lena 512x512 image using SHA-1, SHA-256, and SHA-512.

- ✓ Create a digital signature and verify it.

It is not very efficient to sign a big file using directly a public key algorithm. That is why first we compute the digest of the information to sign. Note that in practice things are a bit more complex. The security provided by this scheme (hashing and then signing directly using RSA) is not the same (is less in fact) than signing directly the whole document with the RSA algorithm.

```
> openssl dgst -<hash_algorithm> -out <digest> <input_file>
```

Where:

- `hash_algorithm` is the hash algorithm used to compute the digest. Among the available algorithm there are: *SHA-1* (option `-sha1` which computes a 160 bits digests), *MD5* (option `-md5`) with 128 bits output length and *RIPEMD160* (option `-ripemd160`) with 160 bits output length.
- `digest` is the file that contains the result of the hash application on `input_file`.
- `input_file` file that contains the data to be hashed.

This command can be used to check the hash values of some archive files like the openssl source code for example.

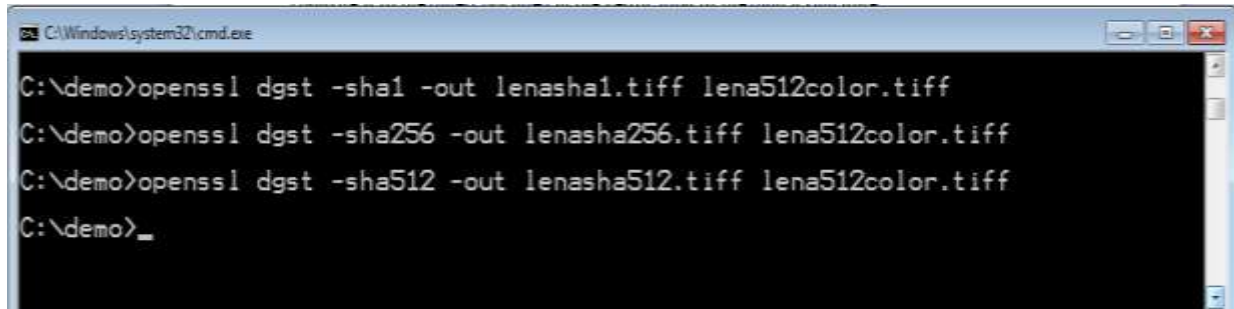
- ✓ Compute the signature of the digest:

```
> openssl rsautl -sign -in <digest> -out <signature> -inkey  
<key>
```

- ✓ Check the validity of the given signature:

```
> openssl rsautl -verify -in <signature> -out <digest> \  
-inkey <key> -pubin
```

-pubin is used like before when the key is the public one, which is natural as we are verifying a signature. To complete the verification, one needs to compute the digest of the input file and to compare it to the digest obtained in the verification of the digital signature.



```
C:\Windows\system32\cmd.exe
C:\demo>openssl dgst -sha1 -out lenasha1.tiff lena512color.tiff
C:\demo>openssl dgst -sha256 -out lenasha256.tiff lena512color.tiff
C:\demo>openssl dgst -sha512 -out lenasha512.tiff lena512color.tiff
C:\demo>_
```

C. PUBLIC KEY ENCRYPTION.

C.1 Performing an RSA encryption on the Lena 512x512 image, using RSA-2048.

- ✓ Create your own small PKI.
- ✓ Configure openssl.cnf

Before starting to create certificates it is necessary to configure a few parameters. That can be done editing the file openssl.cnf which is usually located in the bin directory of OpenSSL. This file looks like this:

[openssl.cnf](#)

```
# OpenSSL example configuration file.
# This is mostly being used for generation of certificate requests.
#

# This definition stops the following lines choking if HOME isn't
# defined.
HOME = .
RANDFILE = $ENV::HOME/.rnd

# Extra OBJECT IDENTIFIER info:
#oid_file = $ENV::HOME/.oid
oid_section = new_oids

# To use this configuration file with the "-extfile" option of the
# "openssl x509" utility, name here the section containing the
# X.509v3 extensions to use:
# extensions =
# (Alternatively, use a configuration file that has only
# X.509v3 extensions in its main [= default] section.)

[ new_oids ]

# We can add new OIDs in here for use by 'ca' and 'req'.
# Add a simple OID like this:
# testoid1=1.2.3.4
```

```

# Or use config file substitution like this:
# testoid2=${testoid1}.5.6

#####
[ ca ]
default_ca = CA_default # The default ca section

#####
[ CA_default ]

dir = "/home/philippe/openssl" # Where everything is kept
certs = $dir/certs             # Where the issued certs are kept
crl_dir = $dir/crl             # Where the issued crl are kept
database = $dir/index.txt      # database index file.
#unique_subject = no           # Set to 'no' to allow creation of
                                # several certificates with same subject.
new_certs_dir = $dir/newcerts  # default place for new certs.

certificate = $dir/cacert.pem  # The CA certificate
serial = $dir/serial           # The current serial number
crlnumber = $dir/crlnumber     # the current crl number
                                # must be commented out to leave a V1 CRL
crl = $dir/crl.pem             # The current CRL
private_key = $dir/private/cakey.pem # The private key
RANDFILE = $dir/private/.rnd   # private random number file

x509_extensions = usr_cert     # The extensions to add to the cert

# Comment out the following two lines for the "traditional"
# (and highly broken) format.
name_opt = ca_default          # Subject Name options
cert_opt = ca_default          # Certificate field options

# Extension copying option: use with caution.
# copy_extensions = copy

# Extensions to add to a CRL. Note: Netscape communicator chokes on V2 CRLs
# so this is commented out by default to leave a V1 CRL.
# crlnumber must also be commented out to leave a V1 CRL.
# crl_extensions = crl_ext

default_days = 365             # how long to certify for
default_crl_days = 30          # how long before next CRL
default_md = sha1              # which md to use.
preserve = no                  # keep passed DN ordering

# A few difference way of specifying how similar the request should look
# For type CA, the listed attributes must be the same, and the optional
# and supplied fields are just that :-)
policy = policy_match

# For the CA policy
[ policy_match ]
countryName = match
stateOrProvinceName = match
organizationName = match
organizationalUnitName = optional
commonName = supplied
emailAddress = optional

....

```

The default `openssl.cnf` file with the `demoCA` directory (also in the `bin` directory of OpenSSL) can also be used, it also contains all the necessary files. Ensure that all the directories are valid ones, and that the private key that will be created in the next section (`cakey.pem`) is well linked. Also check of the presence of a file `.rand` or `.rnd` that will be created with `cakey.pem`. For the certificates database you can create an empty file `index.txt`. Also create a serial file `serial` with the text for example `011E`. `011E` is the serial number for the next certificate.

- ✓ Create a certificate for the PKI that will contain a pair of public / private key. The private key will be used to sign the certificates.

```
> openssl req -new -x509 -keyout cakey.pem -out cacert.pem
```

The pair of keys will be in `cakey.pem` and the certificate (which does NOT contain the private key, only the public) is saved in `cacert.pem`. The private key contained in `cakey.pem` is encrypted with a password. This file should be put in a very secure place (although it is encrypted). `-x509` refers to a standard that defines how information of the certificate is coded.

- ✓ Export the certificate of the PKI in DER format as to be able to load it into your browser.

```
> openssl x509 -in cacert.pem -outform DER -out cacert.der
```

- ✓ Create a user certificate

Now the PKI has got its own pair of keys and certificate, let's suppose a user wants to get a certificate from the PKI. To do so he must create a *certificate request*, that will contain all the information needed for the certificate (name, country, ... and the public key of the user of course). This certificate request is sent to the PKI.

```
> openssl req -new -keyout userkey.pem -out usercert-req.pem
```

Note this command will create the pair of keys and the certificate request. The pair of keys is saved in `userkey.pem` and the certificate request in `usercert-req.pem`.

The PKI is ready for the next step: signing the certificate request to obtain the user's certificate.

```
> openssl ca -in usercert-req.pem -out usercert.pem
Using configuration from /usr/local/bin/openssl/openssl.cnf
Loading 'screen' into random state - done
Enter pass phrase for demoCA/private/cakey.pem:
Check that the request matches the signature
Signature ok
Certificate Details:
    Serial Number: 286 (0x11e)
    Validity
        Not Before: Jan 19 12:52:37 2008 GMT
        Not After : Jan 18 12:52:37 2009 GMT
    Subject:
        countryName           = CL
        stateOrProvinceName   = RM
        organizationName      = littlecryptographer
        commonName            = John Smith
        emailAddress          = jsmith@hello.com
    X509v3 extensions:
        X509v3 Basic Constraints:
            CA:FALSE
        Netscape Comment:
            OpenSSL Generated Certificate
        X509v3 Subject Key Identifier:
            34:F0:61:38:87:68:C7:25:93:86:90:35:32:40:4F:...
        X509v3 Authority Key Identifier:
            keyid:FE:CB:56:0B:28:EB:2A:E9:C7:9C:EA:E5:3A:...

Certificate is to be certified until Jan 18 12:52:37 2009 GMT (365 days)
Sign the certificate? [y/n]:y
```

`usercert.pem` is the public certificate signed by the PKI. If you want to import this certificate into your browser you need to convert it in PKCS12 format:

```
> openssl pkcs12 -export -in usercert.pem -inkey userkey.pem > usercert.p12
```

C.2 Generating an ECDSA signature on the same Lena image using SHA-256.

A private key is a 32-byte number chosen at random, and you know that 32 bytes make for a very big number, as big as 22562256. Such a number is therefore ridiculously hard to guess, assuming it's generated with a very high degree of randomness.

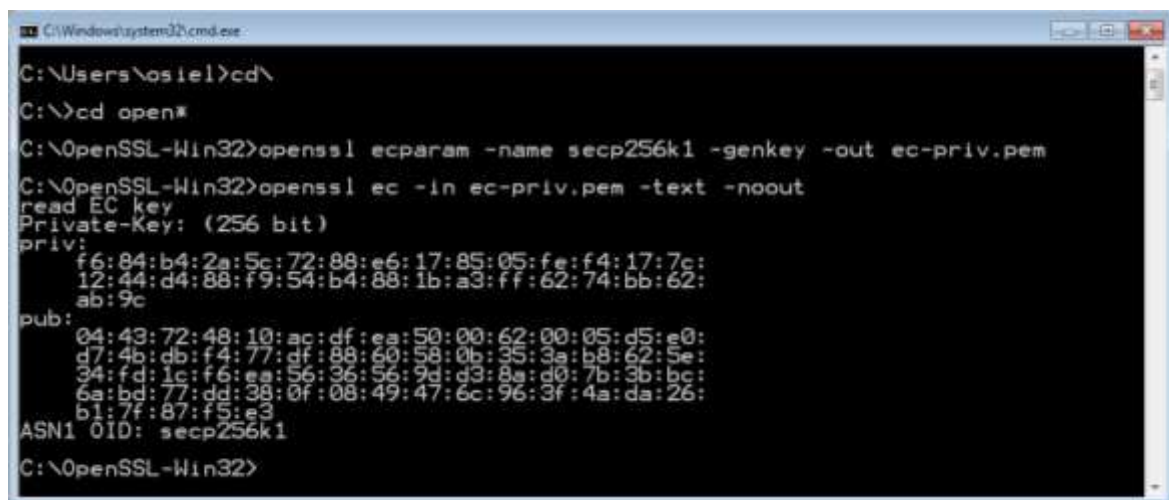
Getting a new, random key is straightforward:

```
$ openssl ecparam -name secp256k1 -genkey -out ec-priv.pem
```

The output file **ec-priv.pem** includes the curve name (secp256k1) and the private key, both encoded base64 with other additional stuff. The file can be quickly decoded to text so that you can see the raw hexes:

```
$ openssl ec -in ec-priv.pem -text -noout
```

Here's what my keypair looks like:



```
C:\Windows\system32\cmd.exe
C:\Users\osiel>cd\
C:\>cd open*
C:\OpenSSL-Win32>openssl ecparam -name secp256k1 -genkey -out ec-priv.pem
C:\OpenSSL-Win32>openssl ec -in ec-priv.pem -text -noout
read EC key
Private-Key: (256 bit)
priv:
  f6:84:b4:2a:5c:72:88:e6:17:85:05:fe:f4:17:7c:
  12:44:d4:88:f9:54:b4:88:1b:a3:ff:62:74:bb:62:
  ab:9c
pub:
  04:43:72:48:10:ac:df:ea:50:00:62:00:05:d5:e0:
  d7:4b:db:f4:77:df:88:60:58:0b:35:3a:b8:62:5e:
  34:fd:1c:f6:ea:56:36:56:9d:d3:8a:d0:7b:3b:bc:
  6a:bd:77:dd:38:0f:08:49:47:6c:96:3f:4a:da:26:
  b1:7f:87:f5:e3
ASN1 OID: secp256k1
C:\OpenSSL-Win32>
```

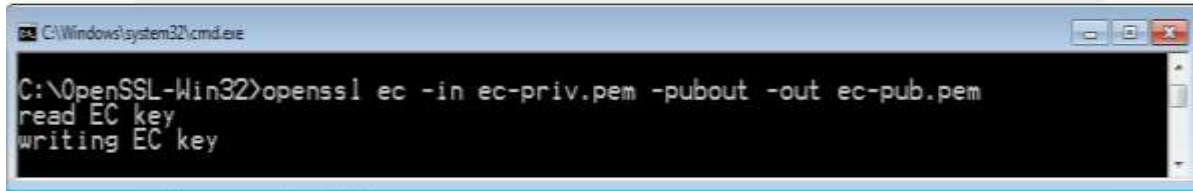
By default, a public key is made of two 32-byte numbers, the so-called *uncompressed* form. The numbers represent the (x,y)(x,y) coordinates of a point on the secp256k1 elliptic curve, which has the following formula:

$$y^2=x^3+7$$

The point location is determined by the private key, while it's unfeasible to infer the private key from the point coordinates. After all, this is what makes EC cryptography secure. Due to its dependent nature, yy can be implied from xx and the curve formula. In fact, the *compressed* form saves space by omitting the yy value.

From the private key, it's possible to take out the public part and store it to an external file I called **ec-pub.pem**:

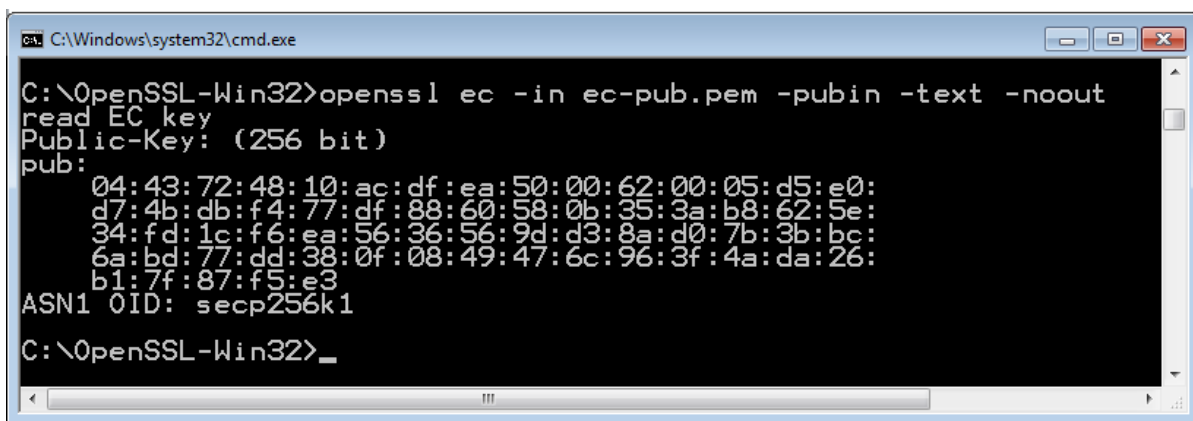
```
$ openssl ec -in ec-priv.pem -pubout -out ec-pub.pem
```



```
C:\Windows\system32\cmd.exe
C:\OpenSSL-Win32>openssl ec -in ec-priv.pem -pubout -out ec-pub.pem
read EC key
writing EC key
```

If we now decode the public key:

```
$ openssl ec -in ec-pub.pem -pubin -text -noout
```



```
C:\Windows\system32\cmd.exe
C:\OpenSSL-Win32>openssl ec -in ec-pub.pem -pubin -text -noout
read EC key
Public-Key: (256 bit)
pub:
    04:43:72:48:10:ac:df:ea:50:00:62:00:05:d5:e0:
    d7:4b:db:f4:77:df:88:60:58:0b:35:3a:b8:62:5e:
    34:fd:1c:f6:ea:56:36:56:9d:d3:8a:d0:7b:3b:bc:
    6a:bd:77:dd:38:0f:08:49:47:6c:96:3f:4a:da:26:
    b1:7f:87:f5:e3
ASN1 OID: secp256k1
C:\OpenSSL-Win32>_
```

the text description will obviously not include the private key:

Generating a keypair from code

The keypair generation task is cumbersome, yet not difficult with the aid of the OpenSSL library. I wrote a helper function in **ec.h** declared as follows:

```
EC_KEY *bbp_ec_new_keypair(const uint8_t *priv_bytes);
```

We create an EC_KEY structure to hold the keypair:

```
key = EC_KEY_new_by_curve_name(NID_secp256k1);
```

Loading the private key is easy, but requires an intermediate step. Before feeding the input `priv_bytes` to the keypair, we need to translate it to a `BIGNUM`, here named `priv`:

```
BN_init(&priv);
BN_bin2bn(priv_bytes, 32, &priv);
EC_KEY_set_private_key(key, &priv);
```

For complex big numbers operations, OpenSSL needs a context, that's why a `BN_CTX` is also created. The public key derivation needs a deeper understanding of EC math, which is not the aim of this series. Basically, we locate a fixed point `GG` on the curve (the *generator*, `group` in the code) and multiply it by the scalar private key `nn`, a virtually irreversible operation in modular arithmetic. The resulting $P=n*GP=n*G$ is a second point, the public key `pub`. Eventually, the public key is loaded into the keypair:

```
ctx = BN_CTX_new();
BN_CTX_start(ctx);
group = EC_KEY_get0_group(key);
pub = EC_POINT_new(group);
EC_POINT_mul(group, pub, &priv, NULL, NULL, ctx);
EC_KEY_set_public_key(key, pub);
```

Now that you're able to generate EC keypairs, the next step is using them to sign and verify messages.

ECDSA signatures

The EC signature algorithm is ECDSA (Elliptic-Curve Digital Signature Algorithm). In ECDSA, all parties involved must agree on a hash function `H`, because we're going to sign `H` (message) rather than the message itself. It's worth noting that only the signing party `S` has access to the private key, while the verifying party `V` holds the corresponding public key to verify `S` signatures. The same private key and public key from the previous section will be used.

Sign

The first step is putting our message into a file, say `ex-message.txt`:

This is a very confidential message whose SHA-256 digest is:

```
45 54 81 3e 91 f3 d5 be
79 0c 7c 60 8f 80 b2 b0
0f 3e a7 75 12 d4 90 39
e9 e3 dc 45 f8 9e 2f 01
```

After that, we sign the SHA-256 digest of the message with the private key:

```
$ openssl dgst -sha256 -sign ec-priv.pem ex-message.txt >ex-
signature.der
```

The `ex-signature.der` file is the message signature in DER format. OpenSSL uses the DER encoding for any binary output (keys, certificates, signatures etc.), but I'll skip the underlying details. There is no need to know the semantic of an ECDSA signature, it is just a simple pair of big numbers $(r,s)(r,s)$.

To display a hex-encoded signature, just add the -hex flag:

```
$ openssl dgst -sha256 -hex -sign ec-priv.pem ex-message.txt
```

For a repeatable output, though, you'd better hexdump the DER file:

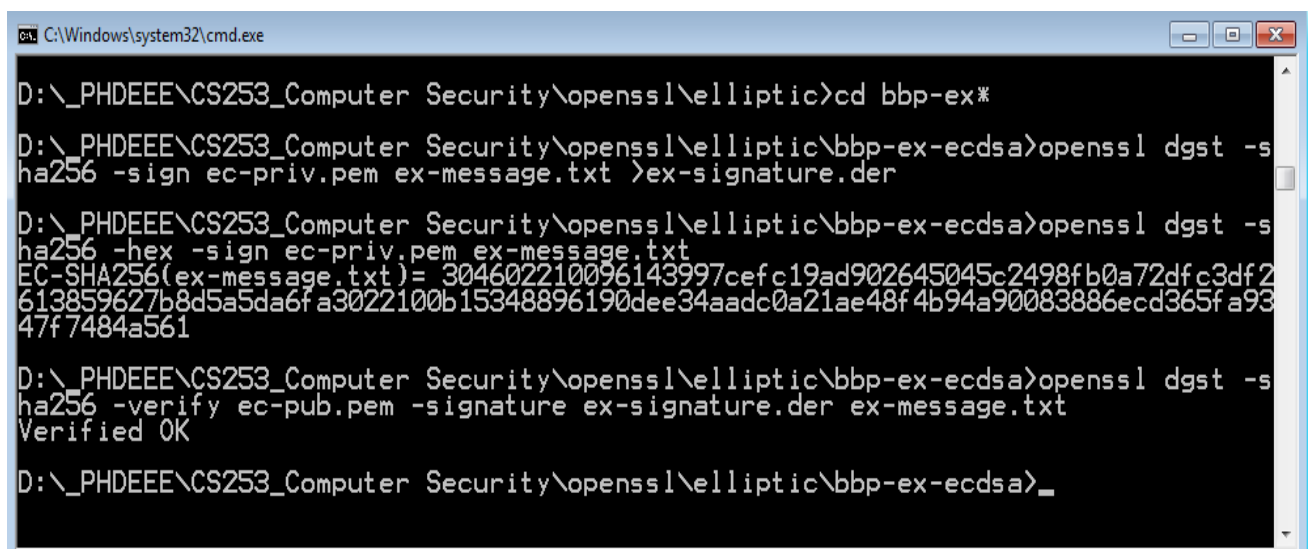
```
$ hexdump ex-signature.der
```

Verify

Whenever an authenticated message is published to the network, the readers expect to find a signature attached. Both files are the input to the verification routine, provided that we received the author's public key in advance:

```
$ openssl dgst -sha256 -verify ec-pub.pem -signature ex-signature.der ex-message.txt
```

If the signature is verified, we're able to state that the message is authentic.



```
C:\Windows\system32\cmd.exe
D:\_PHDEEE\CS253_Computer Security\openssl\elliptic>cd bbp-ex*
D:\_PHDEEE\CS253_Computer Security\openssl\elliptic\bbp-ex-ecdsa>openssl dgst -sha256 -sign ec-priv.pem ex-message.txt >ex-signature.der
D:\_PHDEEE\CS253_Computer Security\openssl\elliptic\bbp-ex-ecdsa>openssl dgst -sha256 -hex -sign ec-priv.pem ex-message.txt
EC-SHA256(ex-message.txt)= 304602210096143997cefc19ad902645045c2498fb0a72dfc3df2613859627b8d5a5da6fa3022100b15348896190dee34aadc0a21ae48f4b94a90083886ecd365fa9347f7484a561
D:\_PHDEEE\CS253_Computer Security\openssl\elliptic\bbp-ex-ecdsa>openssl dgst -sha256 -verify ec-pub.pem -signature ex-signature.der ex-message.txt
Verified OK
D:\_PHDEEE\CS253_Computer Security\openssl\elliptic\bbp-ex-ecdsa>_
```

Code translation

The code below does what we did from the command line in the previous section.

**Sign*

OpenSSL makes the signing operation trivial, look at ex-ecdsa-sign.c:

```
uint8_t priv_bytes[32] = { ... };
const char message[] = "This is a very confidential message\n";

EC_KEY *key;
uint8_t digest[32];
ECDSA_SIG *signature;
uint8_t *der, *der_copy;
```

```

size_t der_len;

...

key = bbp_ec_new_keypair(priv_bytes);
bbp_sha256(digest, (uint8_t *)message, strlen(message));
signature = ECDSA_do_sign(digest, sizeof(digest), key);

```

where ECDSA_SIG is a simple structure holding the (r,s)(r,s) pair described in the previous paragraph:

```

struct {
    BIGNUM *r;
    BIGNUM *s;
} ECDSA_SIG;

```

My test output:

```

r: 2B2B529BDBDC93E78AF7E00228B179918B032D76902F74EF454426F7D06CD0F9
s: 62DDC76451CD04CB567CA5C5E047E8AC41D3D4CF7CB92434D55CB486CCCF6AF2

```

With the i2d_ECDSA_SIG function we also get the DER-encoded signature:

```

der_len = ECDSA_size(key);
der = calloc(der_len, sizeof(uint8_t));
der_copy = der;
i2d_ECDSA_SIG(signature, &der_copy);

```

that in my test is tidily rendered like this:

```

30 44
02 20
2b 2b 52 9b db dc 93 e7
8a f7 e0 02 28 b1 79 91
8b 03 2d 76 90 2f 74 ef
45 44 26 f7 d0 6c d0 f9
02 20
62 dd c7 64 51 cd 04 cb
56 7c a5 c5 e0 47 e8 ac
41 d3 d4 cf 7c b9 24 34
d5 5c b4 86 cc cf 6a f2

```

Verify

Verifying the signature is easy too, here's ex-ecdsa-verify.c:

```

uint8_t pub_bytes[33] = { ... };
uint8_t der_bytes[] = { ... };
const char message[] = "This is a very confidential message\n";

EC_KEY *key;

```

```

const uint8_t *der_bytes_copy;
ECDSA_SIG *signature;
uint8_t digest[32];
int verified;
...
key = bbp_ec_new_pubkey(pub_bytes);
der_bytes_copy = der_bytes;
signature = d2i_ECDSA_SIG(NULL, &der_bytes_copy, sizeof(der_bytes));

```

Since we don't own the private key, we'll have to decode `pub_bytes` into a compressed public key with the following helper from `ec.h`:

```
EC_KEY *bbp_ec_new_pubkey(const uint8_t *pub_bytes, size_t pub_len);
```

On the other hand, `der_bytes` is the DER-encoded signature returned by the signing program. We decode the DER signature into a convenient `ECDSA_SIG` structure and then do the verification against the same SHA-256 message digest:

```

bbp_sha256(digest, (uint8_t *)message, strlen(message));
verified = ECDSA_do_verify(digest, sizeof(digest), signature, key);

```

The `ECDSA_do_verify` function returns:

- 1 if the signature is valid.
- 0 if the signature is not valid.
- 1 for unexpected errors.

The signature decoding can be skipped by using `ECDSA_verify`, which takes a DER-encoded signature directly.

Git Repository:

<https://github.com/Rosiela/EllipticCurve>

References:

- Didier Stevens. Howto: Make Your Own Cert With OpenSSL on Windows
<https://blog.didierstevens.com/category/encryption/> March 30, 2015.
Accessed: May 20, 2017
- [Philippe Camacho](https://users.dcc.uchile.cl/~pcamacho/index.html), An Introduction to the OpenSSL command line tool.
<https://users.dcc.uchile.cl/~pcamacho/index.html>. Accessed: May 20, 2017
- Paul Heinlein . OpenSSL Command-Line HOWTO.
<https://www.madboa.com/geek/openssl/>. Accessed: May 20, 2017.
- Encrypt and decrypt files to public keys via the OpenSSL Command Line.
https://raymii.org/s/tutorials/Encrypt_and_decrypt_files_to_public_keys_via_the_OpenSSL_Command_Line.html. Accessed: May 20, 2017
- Davide De Rosa. A developer-oriented series about Bitcoin.
<http://davidederosa.com/basic-blockchain-programming/> Accessed: May 20, 2017.