Lab Work 3

Data Protection
Master in Cyber Security (UPC) 2022
Jorge L. Villar

Last updated: Nov 14 20:12:52 2022

Contents

1 Key Generation with openssl

The `openssl` tool provides specific commands for key generation of some public key encryption schemes.

The command `genpkey` is used to generate a public / secret keypair.

The way this command is used depends on the selected public key algorithm.

1.1 RSA Keys

```
$ openssl genpkey -algorithm rsa
```

outputs a text encoding of a public key and a secret key for the RSA encryption scheme, with the default parameters (e.g., the size of the modulus). For example, the previous call results in the output

```
-----BEGIN PRIVATE KEY-----
MIICdwIBADANBgkqhkiG9w0BAQEFAASCAmEwggJdAgEAAoGBAKT7B3Vi+hdOPXr5
lIjR3Ao5U4JjbdmQ4hhX5hh/uJwEEA7GkKrDTGOqLTcZZgeXH4nrq5SwoG6O4Mi1
hT6s/FSdpRkl6LFNcjpXPT4GtzPycMKhsGu+rnLiJmuXtEMM1Cw3gutYVa63ygJx
f7YDcJxI2FcAd02jsKA11MP6n4CTAgMBAAECgYBfsU8xMli3PeWBN9SEy5zivT+H
6J4lzNinoAxRd3uf2udpepkcwyzCvkl9pRi+HFTpza13ED/uAKe3IzqHERVGHFMd
qduffQg5DNDymkqnCcJ5yZLe9qn3tj3/EW3Om3Z2Pvn6R9NLFbN8YnTwjWowFmFe
mROdYticID6Ilrv54QJBANqGNGQ2qcDTAeaYeWLY5wnvsm/Apgk3u2qogJB8dp+z
rfey7OAq4B5n+r+OFMZPyqHcs4olUvrCjiEEb/eiL7ECQQDBRh2YpzmW2NBNt+Qy
```

(Generated by lineprocx v2.97)

```
UMrGImRli7GFlp9UKhkRwiOhR1p48mD7yjbSqb7eG0QEVLLN5F02AeALlZDFvTd4
CemDAkEApBc6qDXT6pOITdwY6nztoKx5VSlYhHtxJHo7cEPF385QyDt3XC1V9f8m
b2WOZAvuoPTVbNryIJKPn4NxgIYtQQJAHembJQgkmpsdyhl+4OauK3IhNbIkHHfO
WvlU/fGdEBX6A6QHrJCEYaBR4RA5O65cKg+A+Z3arhBM4r3BOvvVvwJBAJ2A3l0p
TCNwuGbuUVw8KjJ6zHoLro3pTruhgDZvfjvq+ymFmQ2/o6m0ULiJ2XsUqVnpdzYT
WuOKOkYMxNv3GDo=
-----END PRIVATE KEY-----
```

The text output is given in PEM (Privacy-Enhanced Mail) format, but you can also produce a human readable text format by adding the option `-text` to the command line:

```
Private-Key:  (1024 bit)
modulus:
    00:a4:fb:07:75:62:fa:17:4e:3d:7a:f9:94:88:d1:
    dc:0a:39:53:82:63:6d:d9:90:e2:18:57:e6:18:7f:
    b8:9c:04:10:0e:c6:90:aa:c3:4c:63:aa:2d:37:19:
    66:07:97:1f:89:eb:ab:94:b0:a0:6e:8e:e0:c8:b5:
    85:3e:ac:fc:54:9d:a5:19:25:e8:b1:4d:72:3a:57:
    3d:3e:06:b7:33:f2:70:c2:a1:b0:6b:be:ae:72:e2:
    26:6b:97:b4:43:0c:d4:2c:37:82:eb:58:55:ae:b7:
    ca:02:71:7f:b6:03:70:9c:48:d8:57:00:77:4d:a3:
    b0:a0:35:d4:c3:fa:9f:80:93
publicExponent:  65537 (0x10001)
privateExponent:
    5f:b1:4f:31:32:58:b7:3d:e5:81:37:d4:84:cb:9c:
    e2:bd:3f:87:e8:9e:25:cc:d8:a7:a0:0c:51:77:7b:
    9f:da:e7:69:7a:99:1c:c3:2c:c2:be:49:7d:a5:18:
    be:1c:54:e9:cd:ad:77:10:3f:ee:00:a7:b7:23:3a:
    87:11:15:46:1c:53:1d:a9:db:9f:7d:08:39:0c:d0:
    f2:9a:4a:a7:09:c2:79:c9:92:de:f6:a9:f7:b6:3d:
    ff:11:6d:ce:9b:76:76:3e:f9:fa:47:d3:4b:15:b3:
    7c:62:74:f0:8d:6a:30:16:61:5e:99:1d:1d:62:d8:
    9c:20:3e:88:96:bb:f9:e1
prime1:
    00:da:86:34:64:36:a9:c0:d3:01:e6:98:79:62:d8:
    e7:09:ef:b2:6f:c0:a6:09:37:bb:6a:a8:80:90:7c:
    76:9f:b3:ad:f7:b2:ec:e0:2a:e0:1e:67:fa:bf:b4:
    14:c6:4f:ca:a1:dc:b3:8a:25:52:fa:c2:8e:21:04:
    6f:f7:a2:2f:b1
prime2:
    00:c1:46:1d:98:a7:39:96:d8:d0:4d:b7:e4:32:50:
    ca:c6:22:64:65:8b:b1:85:96:9f:54:2a:19:11:c2:
    23:a1:47:5a:78:f2:60:fb:ca:36:d2:a9:be:de:1b:
    44:04:54:b2:cd:e4:5d:36:01:e0:0b:95:90:c5:bd:
    37:78:09:e9:83
exponent1:
    00:a4:17:3a:a8:35:d3:ea:93:88:4d:dc:18:ea:7c:
    ed:a0:ac:79:55:29:58:84:7b:71:24:7a:3b:70:43:
    c5:df:ce:50:c8:3b:77:5c:2d:55:f5:ff:26:6f:65:
    8e:64:0b:ee:a0:f4:d5:6c:da:f2:20:92:8f:9f:83:
    71:80:86:2d:41
exponent2:
    1d:e9:9b:25:08:24:9a:9b:1d:ca:19:7e:e0:e6:ae:
```

```
    2b:72:21:35:b2:24:1c:77:ce:5a:f9:54:fd:f1:9d:
    10:15:fa:03:a4:07:ac:90:84:61:a0:51:e1:10:39:
    3b:ae:5c:2a:0f:80:f9:9d:da:ae:10:4c:e2:bd:c1:
    3a:fb:d5:bf
coefficient:
    00:9d:80:de:5d:29:4c:23:70:b8:66:ee:51:5c:3c:
    2a:32:7a:cc:7a:0b:ae:8d:e9:4e:bb:a1:80:36:6f:
    7e:3b:ea:fb:29:85:99:0d:bf:a3:a9:b4:50:b8:89:
    d9:7b:14:a9:59:e9:77:36:13:5a:e3:8a:3a:46:0c:
    c4:db:f7:18:3a
```

In this version of OpenSSL (1.0.2g), the defaults for RSA are a size of 1024 bits and the public exponent 65537. In newer versions the default RSA key size is 2048 bits. The additional fields named `exponent1`, `exponent2` and `coefficient` are given for optimization of the decryption operation.

You can override the default parameters by providing the desired values via the option `-pkeyopt`. For instance,

```
$ openssl genpkey -algorithm rsa -pkeyopt rsa_keygen_bits:512 -pkeyopt rsa_keygen_pubexp:17
```

will use the (insecure) specified values for the key generation.

1.2 Diffie-Hellman Keys

Other public key cryptosystems, like the Diffie-Hellman key exchange, require a previous generation of some public parameters.

The same command `genpkey` can be used for public parameter generation with the option `-genparam`.

For instance, (the order of the options is important!)

```
$ openssl genpkey -genparam -algorithm dh -pkeyopt dh_paramgen_prime_len:512 -text
```

produces the output

```
-----BEGIN DH PARAMETERS-----
MEYCQQCp/FMOYEOyuktyyNZea1SR7WZ8E78+j5W7qu+Kwqx6xU4Du5Bvoo+9lK0+
eku5tkScHH3TIMcyLLiTDqldWAprAgEC
-----END DH PARAMETERS-----



DH Parameters:  (512 bit)
    prime:
        00:a9:fc:53:0e:60:43:b2:ba:4b:72:c8:d6:5e:6b:
        54:91:ed:66:7c:13:bf:3e:8f:95:bb:aa:ef:8a:c2:
        ac:7a:c5:4e:03:bb:90:6f:a2:8f:bd:94:ad:3e:7a:
        4b:b9:b6:44:9c:1c:7d:d3:20:c7:32:2c:b8:93:0e:
        a9:5d:58:0a:6b
    generator:  2 (0x2)
```

corresponding to the PEM and the plain text output formats. Observe that no public / secret keys are

generated, but only the description of the Diffie-Hellman group. The labels `prime` and `generator` can be different across OpenSSL versions.

You can save the PEM output to a file (e.g., providing the option `-out` filename) and use it for actual key generation. This way, you can reuse the same parameters for several keys (e.g., for the two parties in a Diffie-Hellman key exchange).

Given the DH parameter file `mydhgroup.pem`, you can now generate a keypair with

```
$ openssl genpkey -paramfile mydhgroup.pem -text
```

You don't need to provide the name of the public key algorithm or the size parameters, because they can be inferred from the public parameters file header. The output produced by the above command is:

```
-----BEGIN PRIVATE KEY-----
MIGcAgEAMFMGCSqGSIb3DQEDATBGAkEAqfxTDmBDsrpLcsjWXmtUke1mfBO/Po+V
u6rvisKsesVOA7uQb6KPvZStPnpLubZEnBx90yDHMiy4kw6pXVgKawIBAgRCAkBx
l8GJCiMPoHx/lZ5I9mNWGlbsRF7gc85d+DXlqNLBSKcwK6KYreCvFFT6ASZTGEuB
t+ON4ZtsEyEQMhnxbokb
-----END PRIVATE KEY-----



DH Private-Key:  (512 bit)
    private-key:
        71:97:c1:89:0a:23:0f:a0:7c:7f:95:9e:48:f6:63:
        56:1a:56:ec:44:5e:e0:73:ce:5d:f8:35:e5:a8:d2:
        c1:48:a7:30:2b:a2:98:ad:e0:af:14:54:fa:01:26:
        53:18:4b:81:b7:e3:8d:e1:9b:6c:13:21:10:32:19:
        f1:6e:89:1b
    public-key:
        78:1f:c1:e3:e2:7d:22:8d:ad:0a:e2:dd:91:cd:49:
        e0:94:19:b0:20:4c:cd:9d:be:68:98:84:a5:9b:96:
        5e:c1:46:98:12:c8:e9:ee:f9:d1:48:66:7a:b4:da:
        fa:46:12:f3:a4:ea:1f:a0:a6:a1:34:54:97:33:d5:
        d5:92:2a:21
    prime:
        00:a9:fc:53:0e:60:43:b2:ba:4b:72:c8:d6:5e:6b:
        54:91:ed:66:7c:13:bf:3e:8f:95:bb:aa:ef:8a:c2:
        ac:7a:c5:4e:03:bb:90:6f:a2:8f:bd:94:ad:3e:7a:
        4b:b9:b6:44:9c:1c:7d:d3:20:c7:32:2c:b8:93:0e:
        a9:5d:58:0a:6b
    generator:  2 (0x2)
```

Observe that both the prime and the generator are the same as in the given parameters file.

The prime generated with the previous commands is a safe prime (i.e., is has the form $p = 2q + 1$, where q is also prime). This option makes the parameter generation slow and also impacts the performance of the public key operations for these parameters, without a real security improvement

You can use a faster generation following the X9.42 specification (see RFC2631 for more details), in which $p = kq + 1$ and the prime q can be smaller (e.g., 224 bits long). For that, you need to add the extra option `-pkeyopt dh_paramgen_type:1`, and you can also choose the size q of the DH group with `-pkeyopt dh_paramgen_subprime_len:`value, where value must be one of 160, 224 or 256.

For instance, the parameters generated with

```
$ openssl genpkey -genparam -algorithm dh -pkeyopt dh_paramgen_type:1 -pkeyopt \
$ dh_paramgen_prime_len:512 -pkeyopt dh_paramgen_subprime_len:160 -text
```

are

```
-----BEGIN X9.42 DH PARAMETERS-----
MIGdAkEA6Bq3mgJlZJR7ugkcEice6okyZHj4HHiOlsPGnOEFQWWu4wXqZiH3OLcr
MkBaFIZRlLHPAeMnKPZ1oxW7Ek2Z5wJBAJHBQ2ONsKTeQbeTrVGUGOPYQvFeRoNd
8dHwQRDRHF6tm2ixb/WOqm1xiDffBfduerQlEGx/OLTI/MwMagIIYfYCFQDd9jWt
+nDH56jw49p5ND9bz3OCkQ==
-----END X9.42 DH PARAMETERS-----



DH Parameters:  (512 bit)
    prime:
        00:e8:1a:b7:9a:02:65:64:94:7b:ba:09:1c:12:27:
        1e:ea:89:32:64:78:f8:1c:78:8e:96:c3:c6:9f:41:
        05:41:65:ae:e3:05:ea:66:21:f7:38:b7:2b:32:40:
        5a:14:86:51:94:b1:cf:01:e3:27:28:f6:75:a3:15:
        bb:12:4d:99:e7
    generator:
        00:91:c1:43:6d:0d:b0:a4:de:41:b7:93:ad:51:94:
        1b:43:d8:42:f1:5e:46:83:5d:f1:d1:f0:41:10:d1:
        1c:5e:ad:9b:68:b1:6f:f5:8e:aa:6d:71:88:37:df:
        05:f7:6e:7a:b4:25:10:6c:7f:38:b4:c8:fc:cc:0c:
        6a:02:08:61:f6
    subgroup order:
        00:dd:f6:35:ad:fa:70:c7:e7:a8:f0:e3:da:79:34:
        3f:5b:cf:73:82:91
    recommended-private-length:  160 bits
```

In newer OpenSSl versions (from 3.0 on) the command syntax and the field names in the output are slightly different. The name of the algorithm is now `dhx` instead of `dh`. Then the corresponding command will be

```
$ openssl genpkey -genparam -algorithm dhx -pkeyopt dh_paramgen_prime_len:512 \
$ -pkeyopt dh_paramgen_subprime_len:160 -text
```

You can also use any of the three standarized DH groups (see RFC5114, sections 2.1, 2.2, 2.3) with the option `-pkeyopt dh_rfc5114:`value, where value is one of 1, 2 or 3, as in the example

```
$ openssl genpkey -genparam -algorithm dh -pkeyopt dh_rfc5114:3 -text
```

and again, in newer versions of OpenSSL, you must replace `-algorithm dh` by `-algorithm dhx`.

The key files generated with the faster DH groups (either using X9.42 or RFC5114) have some additional fields, e.g., specifying the group order q. For example, using the first group in RFC5114, we can build the following keyfile:

```
DH Private-Key:  (1024 bit)
    private-key:
        3d:0f:71:1e:47:44:b2:e1:b7:37:50:01:ea:34:c5:
        a5:93:91:71:7f
    public-key:
        7e:ee:7b:7e:7f:83:c8:5e:68:1c:6b:6e:43:49:11:
        4f:c3:1d:9a:8c:6e:3c:62:68:d6:fd:ff:06:bc:39:
        21:4f:ec:5a:64:a4:ed:16:6e:1f:07:e4:e6:83:31:
        79:e8:07:8b:61:e0:07:ce:33:12:0b:ec:d9:5a:51:
        0e:62:1c:68:c7:f4:01:53:f8:5b:b1:0f:2d:aa:31:
        1e:39:50:85:83:ff:76:b1:93:f7:67:e5:5c:af:36:
        6a:e7:bf:78:b1:6e:1d:d3:dc:95:60:b5:9f:a1:d0:
        14:ea:b7:eb:a6:3c:be:31:2d:90:b6:e6:2a:c8:f9:
        08:43:4e:c4:fc:59:c4:1e
    prime:
        00:b1:0b:8f:96:a0:80:e0:1d:de:92:de:5e:ae:5d:
        54:ec:52:c9:9f:bc:fb:06:a3:c6:9a:6a:9d:ca:52:
        d2:3b:61:60:73:e2:86:75:a2:3d:18:98:38:ef:1e:
        2e:e6:52:c0:13:ec:b4:ae:a9:06:11:23:24:97:5c:
        3c:d4:9b:83:bf:ac:cb:dd:7d:90:c4:bd:70:98:48:
        8e:9c:21:9a:73:72:4e:ff:d6:fa:e5:64:47:38:fa:
        a3:1a:4f:f5:5b:cc:c0:a1:51:af:5f:0d:c8:b4:bd:
        45:bf:37:df:36:5c:1a:65:e6:8c:fd:a7:6d:4d:a7:
        08:df:1f:b2:bc:2e:4a:43:71
    generator:
        00:a4:d1:cb:d5:c3:fd:34:12:67:65:a4:42:ef:b9:
        99:05:f8:10:4d:d2:58:ac:50:7f:d6:40:6c:ff:14:
        26:6d:31:26:6f:ea:1e:5c:41:56:4b:77:7e:69:0f:
        55:04:f2:13:16:02:17:b4:b0:1b:88:6a:5e:91:54:
        7f:9e:27:49:f4:d7:fb:d7:d3:b9:a9:2e:e1:90:9d:
        0d:22:63:f8:0a:76:a6:a2:4c:08:7a:09:1f:53:1d:
        bf:0a:01:69:b6:a2:8a:d6:62:a4:d1:8e:73:af:a3:
        2d:77:9d:59:18:d0:8b:c8:85:8f:4d:ce:f9:7c:2a:
        24:85:5e:6e:eb:22:b3:b2:e5
    subgroup order:
        00:f5:18:aa:87:81:a8:df:27:8a:ba:4e:7d:64:b7:
        cb:9d:49:46:23:53
```

or from version 3.0 on:

```
DH Private-Key:  (1024 bit)
    private-key:
        3d:0f:71:1e:47:44:b2:e1:b7:37:50:01:ea:34:c5:
        a5:93:91:71:7f
    public-key:
        7e:ee:7b:7e:7f:83:c8:5e:68:1c:6b:6e:43:49:11:
        4f:c3:1d:9a:8c:6e:3c:62:68:d6:fd:ff:06:bc:39:
        21:4f:ec:5a:64:a4:ed:16:6e:1f:07:e4:e6:83:31:
        79:e8:07:8b:61:e0:07:ce:33:12:0b:ec:d9:5a:51:
        0e:62:1c:68:c7:f4:01:53:f8:5b:b1:0f:2d:aa:31:
        1e:39:50:85:83:ff:76:b1:93:f7:67:e5:5c:af:36:
        6a:e7:bf:78:b1:6e:1d:d3:dc:95:60:b5:9f:a1:d0:
```

```
        14:ea:b7:eb:a6:3c:be:31:2d:90:b6:e6:2a:c8:f9:
        08:43:4e:c4:fc:59:c4:1e
    GROUP: dh_1024_160
```

1.3 Elliptic Curves

The elliptic curve version of Diffie-Hellman key generation is very similar, but it currently supports only known (standarized) elliptic curve groups (that is, you cannot use `openssl` to generate new random elliptic curves with the required security properties). In order to create the parameters file for one of the preimplemented elliptic curve groups (e.g., `P-256`, or equivalently, `prime256v1`), you simply use

```
$ openssl genpkey -genparam -algorithm ec -pkeyopt ec_paramgen_curve:P-256 -pkeyopt \
$ ec_param_enc:named_curve -text
```

producing the output

```
-----BEGIN EC PARAMETERS-----
BggqhkjOPQMBBw==
-----END EC PARAMETERS-----



ECDSA-Parameters:  (256 bit)
ASN1 OID: prime256v1
NIST CURVE: P-256
```

You can also generate the explicit values of the parameters providing the value `explicit` to the parameter `ec_param_enc`, like in

```
$ openssl genpkey -genparam -algorithm ec -pkeyopt ec_paramgen_curve:P-256 -pkeyopt \
$ ec_param_enc:explicit -text
```

and the output is now

```
-----BEGIN EC PARAMETERS-----
MIH3AgEBMCwGByqGSM49AQECIQD/////AAAAAQAAAAAAAAAAAAAAAP//////////
/////zBbBCD/////AAAAAQAAAAAAAAAAAAAAAP///////////////AQgWsY12Ko6
k+ez671VdpiGvGUdBrDMU7D2O848PifSYEsDFQDEnTYIhucEk2pmeOETnSa3gZ9+
kARBBGsX0fLhLEJH+Lzm5WOkQPJ3A32BLeszoPShOUXYmMKWT+NC4v4af5uO5+tK
fA+eFivOM1drMV7Oy7ZAaDe/UfUCIQD/////AAAAAP//////////vOb6racXnoTz
ucrC/GMlUQIBAQ==
-----END EC PARAMETERS-----



ECDSA-Parameters:  (256 bit)
Field Type:  prime-field
Prime:
    00:ff:ff:ff:ff:00:00:00:01:00:00:00:00:00:00:00:
```

```
    00:00:00:00:00:00:ff:ff:ff:ff:ff:ff:ff:ff:ff:
    ff:ff:ff
A:
    00:ff:ff:ff:ff:00:00:00:01:00:00:00:00:00:00:
    00:00:00:00:00:00:ff:ff:ff:ff:ff:ff:ff:ff:ff:
    ff:ff:fc
B:
    5a:c6:35:d8:aa:3a:93:e7:b3:eb:bd:55:76:98:86:
    bc:65:1d:06:b0:cc:53:b0:f6:3b:ce:3c:3e:27:d2:
    60:4b
Generator (uncompressed):
    04:6b:17:d1:f2:e1:2c:42:47:f8:bc:e6:e5:63:a4:
    40:f2:77:03:7d:81:2d:eb:33:a0:f4:a1:39:45:d8:
    98:c2:96:4f:e3:42:e2:fe:1a:7f:9b:8e:e7:eb:4a:
    7c:0f:9e:16:2b:ce:33:57:6b:31:5e:ce:cb:b6:40:
    68:37:bf:51:f5
Order:
    00:ff:ff:ff:ff:00:00:00:00:ff:ff:ff:ff:ff:ff:
    ff:ff:bc:e6:fa:ad:a7:17:9e:84:f3:b9:ca:c2:fc:
    63:25:51
Cofactor:    1 (0x1)
Seed:
    c4:9d:36:08:86:e7:04:93:6a:66:78:e1:13:9d:26:
    b7:81:9f:7e:90
```

where you can see the values of the prime defining the finite field, the coefficients a, b of the curve, the generator (both the x- and y-coordinates of the point are packed into a single string), the order or size q of the generated group, and other parameters (cofactor and seed).

You can obtain a list of implemented elliptic curves with

```
$ openssl ecparam -list_curves
```

Finally, the generation of a keypair from a given parameters file (e.g., `P-128_ecgroup.pem`) is exactly as above:

```
$ openssl genpkey -paramfile P-128_ecgroup.pem -text
```

producing the output

```
-----BEGIN PRIVATE KEY-----
MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgohhdkTqmahz8bwgh
BbDhmVyaTT4K+f4qatP6/oF8pb+hRANCAAQhmFI+84JDC+eHPmsbDl7qjZlAFDHM
9263g2FZjEs/zZ5W8T6ipYhu+krsX+j8/our09tWrV+5VOL1jkQtcPOb
-----END PRIVATE KEY-----



Private-Key:  (256 bit)
priv:
    00:a2:18:5d:91:3a:a6:6a:1c:fc:6f:08:21:05:b0:
    e1:99:5c:9a:4d:3e:0a:f9:fe:2a:6a:d3:fa:fe:81:
```

```
    7c:a5:bf
pub:
    04:21:98:52:3e:f3:82:43:0b:e7:87:3e:6b:1b:0e:
    5e:ea:8d:99:40:14:31:cc:f7:6e:b7:83:61:59:8c:
    4b:3f:cd:9e:56:f1:3e:a2:a5:88:6e:fa:4a:ec:5f:
    e8:fc:fe:8b:ab:d3:db:56:ad:5f:b9:57:42:f5:8e:
    44:2d:70:f3:9b
ASN1 OID: prime256v1
NIST CURVE: P-256
```

where the secret key is an integer modulo the size of the group q, and the public key is a point in the elliptic curve (again, the two coordinates of the point are packed into a single string).

1.4 File Formats

In the previous examples you can see the need of packing different mathematical objects together in a single file, and this requires the use of some formatting rules. Along with the human readable text output, OpenSSL uses a printable compressed format PEM and a binary format DER to store keys, ciphertext, signatures and certificates, among other binary objects.

The PEM format is a printable version (using the Base64 printable encoding) of the binary DER format, with additional headers and footers, like for instance:

```
-----BEGIN PRIVATE KEY-----
                ...
 (base64 encoded binary DER content)
                ...
-----END PRIVATE KEY-----
```

DER format is basically an unambiguous way to pack a sequence of objects, and each individual object has three parts: type specification, length of the data, and the data itself. For instance, an integer is encoded by as a byte sequence starting with the type identifier `0x02` followed by the number of bytes used in the binary representation of the integer, and then the binary representation itself. The predefined types in DER include booleans, integers, bit strings, octet strings among other types, and aggregations of them.

You can use the command `asn1parse` of `openssl` to look inside a DER or a PEM file and to retrieve all individual entities stored into them. In some `openssl` commands the user can choose the specific output format (DER or PEM) to store keys, ciphertexts, certificates and other binary objects. For instance, the option `-outform` format, where format is one of `der` or `pem`, used in conjunction with `openssl genpkey` selects the format used to store the generated keypair.

1.5 Public Key Extraction

The previous commands show how to generate public / secret keypairs and store them into files. However, in the public key cryptography setting, public keys must be transmitted without revealing the corresponding secret keys. In OpenSSL one can extract the public part of the generated keypair and store it into a separate file. The command

```
$ openssl pkey -in private.pem -pubout -out public.pem
```

reads the keypair from the file `private.pem`, probably generated with `openssl genpkey`, and extracts and

stores the public key into the file `public.pem`. Default input and output formats are PEM, but you can specify DER format in either input or output files with the options `-inform der` and `-outform der`.

Now, the resulting public key can be used in encryption or in signature verification, while the private key file is required for decryption and for signature generation. Private key files can be generated in a protected way, by encrypting them with a symmetric key encryption scheme in the same call to `openssl genpkey`. See the `openssl` documentation for more details.

2 Public Key Encryption and Decryption

Form the keypairs generated as explained in the previous section one can perform public key encryption and decryption.

For the (plain) RSA public key encryption scheme the use of OpenSSL in the command line is quite easy, with the `openssl pkeyutl` command used in conjunction with the `-encrypt` and `-decrypt` options.

The main problem is that a public key encryption only allows encrypting messages with a very precise format (e.g., in RSA a message must be the binary encoding of an integer between 1 and $n-1$, where n is the public modulus). This means that, in practice, we would need a padding scheme, and possibly a chaining mode for long messages.

From a practical point of view, public key encryption schemes are only used to encrypt ephemeral keys to be used in symmetric encryption primitives (like encryption or MAC). Thus, in OpenSSL you can use

```
$ openssl pkeyutl -pubin -inkey myrsapubkey.pem -in ephemeralkey.bin -encrypt -pkeyopt \
$ rsa_padding_mode:oaep -out ciphertext.bin
```

to encrypt the short file `ephemeralkey.bin` with the RSA key contained in `myrsapubkey.pem` using a specific padding scheme like OAEP (see the documentation of OpenSSL for the implemented RSA padding algorithms), and store the encrypted data in `ciphertext.bin`.

There exist a maximum length for the message to be encrypted (the maximum length could be smaller than the length of n, because the padding string takes some minimal space). Normally, short messages can be encrypted successfully, except if you specify `-pkeyopt rsa_padding_mode:none`, in which only messages with exactly the same binary length of n, and not greater than n as an integer, can be encrypted.

Decryption is performed in a similar way (you also have to specify the same padding scheme used in encryption), but now you provide the private key instead of the public one, and use the option `-decrypt` instead of `-encrypt`.

```
$ openssl pkeyutl -inkey myrsaprivkey.pem -in ciphertext.bin -decrypt -pkeyopt \
$ rsa_padding_mode:oaep -out decrypted.bin
```

3 Diffie-Hellman Key Agreement

The generation of the common key in the Diffie-Hellman Key Agreement requires one secret key (say, Alice's secret key) and one public key (sat, Bob's public key). Given the files (generated as above) `alice_priv.pem` and `bob_publ.pem`, Alice can generate the common secret with

```
$ openssl pkeyutl -inkey alice_priv.pem -peerkey bob_publ.pem -derive -out common.bin
```

Similarly, Bob can execute

```
$ openssl pkeyutl -inkey bob_priv.pem -peerkey alice_publ.pem -derive -out common.bin
```

to obtain the same result.

According to the specification of the Diffie-Hellman Key Agreement protocol, the two keypairs must be produced with the same group (i.e., the same parameters file). If the group corresponds to the normal (i.e., non elliptic curve based) Diffie-Hellman Agreement, then the common secret is the binary representation of $g^{s_A s_B}$, where s_A and s_B are the secret keys of the two parties, and g is the group generator. Otherwise, in the elliptic curve based protocol, the common secret is the binary representation of the x-coordinate of the point $(s_A s_B)G$, where G is the base point (generator) of the subgroup.

The common secret must never be directly used as a symmetric key. Instead, it must be hashed or transformed with a secure key derivation function. For instance, you can pipe the previous generation commands and the hash computation with

```
$ openssl pkeyutl -inkey alice_priv.pem -peerkey bob_publ.pem -derive | openssl dgst \
$ -sha256 -binary > commonkey.bin
```

which produces a secure shared 256 bit symmetric key, ready to be used with any encryption or MAC operation.

4 Practical Work: Hashed ElGamal + Hybrid Encryption + HMAC

As a practical exercise you can implement a version of Hashed ElGamal encryption scheme combined with a symmetric encryption scheme and a MAC.

We use a normal DH group, with a size of 256 bits. The common secret resulting from a Diffie-Hellman agreement is 256 bits long. We use SHA256 to obtain a 256 bits symmetric key, that is splitted into two 128 bits keys: one, k_1, for AES-128-CBC encryption and the other, k_2, for SHA256-HMAC.

Before encrypting a file, the recipient must have been generated a parameter file for the DH group and a long-term Diffie Hellman keypair, and then she must have send us the corresponding public key. Namely, she needs to perform the following steps:

- Generate the DH group, or choose one of the predefined standard groups. For instance, she chooses the third group in RFC5114 and generates the corresponding parameters file with

  ```
  $ openssl genpkey -genparam -algorithm dh -pkeyopt dh_rfc5114:3 -out param.pem
  ```

  (from version 3.0 on, you need to replace `-algorithm dh` with `-algorithm dhx`)

- Generate the long-term keypair with

  ```
  $ openssl genpkey -paramfile param.pem -out alice_pkey.pem
  ```

- Extract the long-term public key with

  ```
  $ openssl pkey -in alice_pkey.pem -pubout -out alice_pubkey.pem
  ```

- Publish the files `param.pem` and `alice_pubkey.pem`. Actually, she would need to obtain a certificate for this public key form a certification authority to prevent impersonation attacks.

Now, the encryption operation consists of:

- Compute an ephemeral Diffie-Hellman keypair $r, a = g^r$ with `openssl genpkey` using the file `param.pem` published by the recipient, and store it in the file `ephpkey.pem`.
- Generate the corresponding ephemeral public key file (in PEM format) with `openssl pkey` and store it in the file `ephpubkey.pem`.
- Derive a common secret from the secret ephemeral key r, contained in `ephpkey.pem`, and the long-term public key of the recipient, contained in `alice_pubkey.pem`, with `openssl pkeyutl` with the `-derive` option.
- Apply SHA256 to the common secret with `openssl dgst` (see the previous practical work), and split it into two halves to obtain k_1 and k_2. Starting from the 32 bytes long binary file with the hash value, you can use `head -c 16` and `tail -c 16` to extract the first and the last 16 bytes.
- Encrypt the desired file with AES-128-CBC using key k_1, with `openssl enc -aes-128-cbc` and store the result in the file `ciphertext.bin` (see previous practical works). Some useful tools helps converting binary files into the plain hexadecimal representation, like `xxd -p`. You would need to provide an iv for the encryption operation. You can generate a random one with `openssl rand 16` and store it in the file `iv.bin`.
- Use key k_2 to compute the SHA256-HMAC tag of the concatenation of `iv.bin` and `ciphertext.bin` with `openssl dgst -hmac -sha256` to obtain the binary file `tag.bin` (again, you can find the details in the previous practical work).
- The final ciphertext consists of the four files `ephpubkey.pem`, `iv.bin` , `ciphertext.bin` and `tag.bin`.

Ideally, a single file in PEM or DER format joining the four would be a better option. You can do that by concatenating the printable (base64 encoded) files with adequate headers and footers. For instance, you can start from the file `ephpubkey.pem` that already is in PEM format, then concatenate it with the base64 encoding of `ciphertext.bin` , `iv.bin` and `tag.bin` with some PEM style headers and footers added:

```
   (put here the file ephpubkey.pem as it is)
   (it already contains its own header and footer lines)
-----BEGIN AES-128-CBC IV-----
   (put here the 64bit encoding of file iv.bin)
-----END AES-128-CBC IV-----
-----BEGIN AES-128-CBC CIPHERTEXT-----
   (put here the 64bit encoding of file ciphertext.bin)
-----END AES-128-CBC CIPHERTEXT-----
-----BEGIN SHA256-HMAC TAG-----
   (put here the 64bit encoding of file tag.bin)
-----END SHA256-HMAC TAG-----
```

To convert files between binary and base64 encoding you can use the special OpenSSL pseudocipher in `openssl enc` with the `-a` option. In the encryption direction the input is interpreted as a binary file and the output is the corresponding printable base64 encoding. In the decryption direction (with the `-d` option), the conversion is done in the opposite way.

Decryption can be performed with a similar procedure:

- Parse the ciphertext file and obtain the four components `ephpubkey.pem`, `iv.bin` , `ciphertext.bin` and `tag.bin`.

- Use files `alice_pkey.pem` and `ephpubkey.pem` to recover the common secret with `openssl pkeyutl -derive`.
- Apply SHA256 to the previous result and split the value into the two keys k_1 and k_2.
- Recompute the SHA256-HMAC from the concatenation of the files `iv.bin` and `ciphertext.bin` using the key k_2. If the result is different from the file `tag.bin`, then abort the decryption operation and report the error.
- Use `openssl enc -d -aes-128-cbc` with the key k_1 and the iv given in `iv.bin` to decrypt the file `ciphertext.bin` and recover the plaintext.

In a complete system you would provide three scripts: one for parameter and key generation, another for encryption and a third one for decryption.
