Security in Networked Computer Systems
OpenSSL Lab Session #4

Pericle Perazzo
pericle.perazzo@iet.unipi.it
http://www.iet.unipi.it/p.perazzo/teaching/

1st April 2015
Asymmetric Encryption
An asymmetric cryptosystem uses two keys, one of which is private, the other public. It usually provides for four operations (apart from key generation): public encryption ($E_{k_{pub}}$), private decryption ($E^{-1}_{k_{prv}}$), private encryption ($E_{k_{prv}}$), public decryption ($E^{-1}_{k_{pub}}$). The public encryption is undone by the private decryption. These two operations are used in the *digital envelope* technique. The private encryption is undone by the public decryption. These two operations are used in the *digital signature* technique.
An asymmetric key is not a simple string of bits like a symmetric key, but it has an internal structure. This slide shows how an RSA public/private key is internally represented in OpenSSL. The first two BIGNUM's represent the public key: the modulus \( n \), and the public exponent \( e \). All the BIGNUM's together represent the private key, in particular the private exponent \( d \).

We will not deal with this data structure, as we will use the high-level OpenSSL API (\#include<openssl/evp.h>).
Asymmetric Cryptography

- RSA (Rivest-Shamir-Adleman):
  - Based on NP-hardness of the factorization problem,
  - Quite simple math,
  - Very famous,
  - TOP SECRET security requires very long keys (7680 bits), which are very inefficient.

- EC (Elliptic Curve):
  - Based on NP-hardness of the discrete logarithm problem,
  - Not-so-simple to implement,
  - Reaches the same security of RSA with:
    - shorter keys,
    - more efficiency in key generation,
    - more efficiency in encryption operations,
  - TOP SECRET security requires short keys (384 bits), which are efficient.

The most famous and widespread asymmetric cryptosystems are RSA and EC. RSA (Rivest-Shamir-Adleman, from the names of its inventors) is the oldest and most famous one. It is based on the NP-hardness of the factorization problem. RSA is very famous because it is quite simple to understand and implement. It is widespread in many applications.

However, to obtain high levels of security it requires very long keys (the length of an RSA key is given by the number of bits of the modulus), and the encryption/decryption operations are inefficient. RSA cryptosystem is nowadays technologically obsolete, surpassed by Elliptic Curve cryptography (EC). EC gives the same security of RSA with far shorter keys. It is based on the NP-hardness of the discrete logarithm problem. An inefficient 7680-bit RSA key is equivalent to an efficient 384-bit EC key.
This table shows the security equivalence between RSA and EC keys and the correspondent effective strength, as reported by SECG. SECG is an industry consortium to develop cryptography standards.

It is not straightforward to determine the effective strength of an RSA key, since the complexity of the factorization problem is not easy to compute. A heuristic formula (elaborated from RFC3766) is:

$$\text{strength} = -5.64 + 2.77 \times \text{cubrt}(\text{len}\times\ln(2)\times(\ln(\text{len}\times\ln(2)))^2),$$

where “len” is the length of the key (i.e. of the modulus) in bits, “cubrt()” indicates the cube root, “\ln()” indicates the natural logarithm, and “strength” is the effective strength (in bits). This formula applies for Diffie-Hellman as well, to compute the effective strength given the modulus' length in bits.

The effective strength of an EC key is always half the key length (e.g. 160-bit keys give 80-bit strength). This is because the best known algorithms for solving discrete logarithm run in $2^{(\text{len}/2)}$ time, where “len” is the length of the key in bits.

When creating an EC key pair, it is necessary to specify the curve name, which comprises also the length of the key. Common curves are P-256 (named “prime256v1” in OpenSSL) and P-384 (named “secp384r1” in OpenSSL).
**Digital Envelope**

- Digital envelope encrypts a message in such a way that only who knows a particular private key can open it (no pre-shared secrets).
- A simple method is to encrypt the whole message with the public key of the recipient.
- A better method is to encrypt the message with a symmetric key, and then encrypt the symmetric key with the public key of the recipient.

The digital envelope technique encrypts a message in such a way that only who knows a particular private key can decrypt it. In contrast to symmetric encryption, no pre-shared secret is needed.

A straightforward way to do that is to encrypt the whole message with the public key of the recipient. This is very inefficient, because symmetric encryption is extremely slow compared to symmetric one. A better solution is to encrypt the whole message with a randomly generated symmetric key, and then encrypt only the symmetric key with the public key.
Digital envelope can also be *multi-addressed*, in case we want to send the same confidential message to several recipients. This is done by encrypting the symmetric key with the public key of each recipient.
Key Pair Generation

- Tools for RSA keys generation:
  - `openssl genrsa -out rsa_privkey.pem 1024`
    Generates an RSA private key.
  - `openssl rsa -pubout -in rsa_privkey.pem -out rsa_pubkey.pem`
    Extracts the public key from an RSA private key.

- Tools for EC keys generation:
  - `openssl ecparam -genkey -name prime256v1 -out ec_privkey.pem`
    Generates an EC private key.
  - `openssl ec -pubout -in ec_privkey.pem -out ec_pubkey.pem`
    Extracts the public key from an EC private key.

These are the OpenSSL command-line tools to create a private key and to extract the public key from a private key (both RSA and EC). All the keys are saved in PEM-format files.
PEM Format

- PEM (Privacy-Enhanced Mail) is a textual standard for storing cryptographic material.
  - Public RSA/EC keys,
  - Private RSA/EC keys,
  - Diffie-Hellman parameters,
  - Certificates,
  - Etc.

- Example of 1024-bit RSA public key in PEM format:

```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDerpZiO7yI0Gp7i+EathC74vXv
hEr18c3isbVc0oMO/06cpBZ+8+kvMSOxSrxxz12CBDRkybobO+0IlPz2PzXx1G1
qfpM+dmulZnaXRYYh/Uu9UZL5VLq2rMyQyvcZ1hA6gNf0w/5M9me97heyy4gogLJaq
zx52ol6c/4hpDwQecQIDAQAB
-----END PUBLIC KEY-----
```

PEM (Privacy-Enhanced Mail) is a 1993 IETF standard for securing eMail communications using asymmetric cryptography. It became obsolete once PGP has been published, but the correspondent file format became widespread. The PEM format is a textual one, in which cryptographic quantities are surrounded by “tags”, for example “-----BEGIN PUBLIC KEY-----”, “-----END PUBLIC KEY-----”. It can contain public or private keys (both RSA and EC), digital certificates, Diffie-Hellman parameters, and so on. Another common format is DER, which is a binary format.
OpenSSL API for Asymmetric Cryptography

- `#include <openssl/evp.h>`
  High-level OpenSSL API functions (among which EVP_PKEY)
- **EVP_PKEY (data structure)**
  It represents a public key or a private key (both RSA and EC cryptosystems)
- **EVP_PKEY* EVP_PKEY_new();**
  Allocates an EVP_PKEY.
- **void EVP_PKEY_free(EVP_PKEY* key);**
  Deallocates an EVP_PKEY.
- **int EVP_PKEY_size(EVP_PKEY* key);**
  Returns the maximum size of an encrypted symmetric key (useful for allocating buffers).

An EVP_PKEY data structure represents a private or public key (both RSA or EC). These API functions allocate and deallocate an EVP_PKEY data structure.
OpenSSL API for PEM format

- **#include <openssl/pem.h>**
  API functions for reading/writing PEM-format files.

- **EVP_PKEY* PEM_read_PrivateKey(FILE* fp, NULL, NULL, NULL);**
  Allocates a private key and loads it from a PEM file (RSA or EC).
  - `fp` → File where to read (opened with fopen())
  - It returns the EVP_PKEY structure (or NULL if error).

- **EVP_PKEY* PEM_read_PUBKEY(FILE* fp, NULL, NULL, NULL);**
  Allocates a public key and loads it from a PEM file (RSA or EC).
  - `fp` → File where to read (opened with fopen())
  - It returns the EVP_PKEY structure (or NULL if error).

These API functions allocate and load a public or private key (both RSA and EC) from a PEM-format file.
OpenSSL API for Digital Envelope

- **int EVP_SealInit(EVP_CIPHER_CTX* ctx, const EVP_CIPHER* type, unsigned char** ek, int* ekl, unsigned char* iv, EVP_PKEY** pubk, int npubk);**

  Initializes a context for (multi-addressed) digital envelope. It also generates
  the symmetric key for encryption and a random IV (the PRNG must be
  seeded).
  - **ctx** (input/output) → The context for digital envelope.
  - **type** (input) → The symmetric cipher to use.
  - **ek[0]** (output) → The encrypted symmetric key. In case of multi-addressed
    envelope, the other encrypted symmetric keys are stored in **ek[1]**, **ek[2]**, etc.
  - **ekl** (output) → The length of the encrypted symmetric key(s).
  - **iv** (output) → The generated initialization vector.
  - **pubk[0]** (input) → The public key. In case of multi-addressed envelope, the
    other public keys are **pubk[1]**, **pubk[2]**, etc.
  - **npubk** (input) → The number of public keys (1 if single-address envelope, >1 if
    multi-address envelope).
  - Returns 0 on error, non-0 on success.

This API function initializes a context for (multi-addressed) digital envelope. It
takes as input one (or more) public key(s) and returns as output the encrypted text
and the encrypted symmetric key. It contextually generates a random symmetric key
and an initialization vector, so the PRNG must be seeded properly. The buffer
**ek[0]** must accommodate at least `EVP_PKEY_size(pubk[0])` bytes. The buffer
**iv** must accommodate at least `EVP_CIPHER_iv_length(type)` bytes.

Remember that the cipher context must be previously allocated with `malloc()`
and **EVP_CIPHER_CTX_init()**, and finally deallocated with
**EVP_CIPHER_CTX_cleanup()** and `free()`.

Note: **EVP_SealInit()** and all the OpenSSL API functions for digital
envelope support ONLY RSA cryptosystem. Although digital envelope technique
based on EC is technologically possible (cfr. the standard ECIES: Elliptic-Curve
Integrated Encryption Scheme), it is NOT implemented by OpenSSL (version
1.0.1k).
OpenSSL API for Digital Envelope

- `int EVP_SealUpdate(EVP_CIPHER_CTX* ctx, unsigned char* out, int* outl, unsigned char* in, int inl);`
  Updates a context for digital envelope.
  - `ctx` (input/output) → The context.
  - `out`, `outl` (output) → Output fragment.
  - `in, inl` (input) → Input fragment.
  - Returns 0 on error, non-0 on success.
- `int EVP_SealFinal(EVP_CIPHER_CTX* ctx, unsigned char* out, int* outl);`
  Finalizes a context for digital envelope.
  - `ctx` (input/output) → The context.
  - `out, outl` (output) → Final output fragment.
  - Returns 0 on error, non-0 on success.

These API functions update and finalize a digital envelope context. They act in a similar manner to `EVP_EncryptUpdate()` and `EVP_EncryptFinal()`. 
OpenSSL API for Digital Envelope

- int EVP_OpenInit (EVP_CIPHER_CTX* ctx, EVP_CIPHER* type, unsigned char* ek, int ekl, unsigned char* iv, EVP_PKEY* priv);

Initializes a context for digital envelope decryption.
- ctx (input/output) → The context for digital envelope decryption.
- type (input) → The symmetric cipher to use.
- ek (input) → The encrypted symmetric key.
- ekl (input) → The length of the encrypted symmetric key.
- iv (input) → The initialization vector (or NULL).
- priv (input) → The private key.
- Returns 0 on error, non-0 on success.

This API function initializes a context for envelope decryption. It takes as input a private key and an encrypted symmetric key.

Remember that the cipher context must be previously allocated with malloc() and EVP_CIPHER_CTX_init(), and finally deallocated with EVP_CIPHER_CTX_cleanup() and free().
OpenSSL API for Digital Envelope

- **int EVP_OpenUpdate(EVP_CIPHER_CTX* ctx, unsigned char* out, int* outl, unsigned char* in, int inl);**
  
  Updates a context for digital envelope decryption.
  
  - `ctx` (input/output) → The context.
  - `out, outl` (output) → Output fragment.
  - `in, inl` (input) → Input fragment.
  - Returns 0 on error, non-0 on success.

- **int EVP_OpenFinal(EVP_CIPHER_CTX* ctx, unsigned char* out, int* outl);**
  
  Finalizes a context for digital envelope decryption.
  
  - `ctx` (input/output) → The context.
  - `out, outl` (output) → Final output fragment.
  - Returns 0 on error, non-0 on success.

These API functions update and finalize an envelope decryption context. They act in a similar manner to `EVP_DecryptUpdate()` and `EVP_DecryptFinal()`. 
Final Exercise

- On the server, generate a pair of 1024-bit RSA keys, by using OpenSSL command-line tools.
- Give the public key to the client.
- The client reads a file, envelopes it with the server’s public key (DES in CBC), and sends to the server:
  - the encrypted symmetric key,
  - the initialization vector,
  - the ciphertext.
- The server decrypts the message and saves it on a local file.