Security in practice: Cryptography exercises with OpenSSL library

Diana Berbecaru
diana.berbecaru@polito.it

OpenSSL

- SSLeay: developed in 1995 by Eric A. Young and Tim J. Hudson; from 1998 takes the name OpenSSL (0.9.1c).
  - current (stable) version: 1.0.0d (released 08 Feb 2011)
- downloadable (sources and binaries) from: http://www.openssl.org/source/
- opensource library, composed of two libraries:
  - library of crypto functions (libcrypto)
  - library used for the management of SSL protocol (libssl)
- the main characteristic of this library is the complete implementation of SSLv2, SSLv3 and TLSv1 protocols
- other crypto libraries: Crypto++, Cryptlib, BouncyCastle...

Documentation

- not very much, but the situation is getting better:
  - the most updated: http://www.openssl.org/docs/

- books and articles:
- mailing list: http://www.openssl.org/support/
- the code of demo applications is distributed with the library itself !!!
- the file openssl.txt in the directory doc

Useful books

- Network Security with OpenSSL
- SSL and TLS: Designing and Building Secure Systems

Installing OpenSSL (1)

- typically, by downloading the version from the official web site (http://www.openssl.org/).
  - for obvious security reasons it is always advisable to get the last available openssl version, to update it periodically and to compile it independently on your machine; some distributions of binary files, typically precompiled, are not provided by the developers of OpenSSL but by third parties (for example Win32 OpenSSL disponibile available at : http://www.alproweb.com/products/Win32OpenSSL.html)
  - on Win platform it is possible to install it together with Cygwin Linux-like environment for Windows:
    - http://www.cygwin.com

Installing OpenSSL – Setup of Cygwin

© Diana Berbecaru - Politecnico di Torino (2006-2012)
Overview of OpenSSL features

- symmetric block algorithms: AES (from OpenSSL version 0.9.7), DES, 3DES, DESX, CAST, RC2, RC5, IDEA, Blowfish, in modes CBC, CFB, ECB and OFB; for each cipher the default mode is CBC
- symmetric algorithms of type stream: RC4
- hash algorithms: MD2, MD4, MD5, SHA-1, RIPEMD 160, MDC2
- asymmetric algorithms: RSA, DSA, DH, ECC
- authentication: HMAC
- digital certificates and SSL: X509, X509v3, SSLv3, TLSv1
- Input/Output formats: asn1, bio, evp, pem, pkcs7, pkcs12
- internal functions: bn, buffer, lhash, object, stack

Command-line interface of OpenSSL

- OpenSSL can be used by application developers to include cryptographic support into their applications ...
- ...but it can be used also as command-line security tool (called openssl on Linux and openssl.exe on Windows)
- the tool allows users to use the characteristics of the library from the command line, e.g. calculate the hash of a message, or encrypt/decrypt – sign/verify with symmetric and asymmetric cryptography respectively.
- used in "interactive" or "batch" modes

Interactive Mode

- by executing openssl (with no option), the tool enters in "interactive" mode; a prompt indicates that the program is ready to process standard openssl commands:

  $ openssl
  Openssl > standard openssl commands

- when the execution of a command is completed, the prompt reappears, indicating that the tool is ready to execute another openssl command
- you can go out from the command-line tool by executing: quit

Batch Mode

- by executing openssl (with options); similar to the "interactive" mode, except that for each openssl command it must be added openssl in the front

  $ openssl standard openssl commands

- syntax of standard openssl commands

  name command [options]

  in general (with some exception), the order of options is not important

Standard openssl commands (1)

- asn1parse
- parses a ASN.1 sequence
- base64
  - encodes/decodes in/from Base 64 format
- ca
  - management of a test Certification Authority (CA)
- ciphers
  - description of a cipher suite
- crl
  - management of Certificate Revocation List (CRL)
- crl2pkcs7
  - conversion of CRL to PKCS#7
### Standard openssl commands (2)

- **dgst**
  - computation of a message digest
- **dh**
  - management of Diffie-Hellman parameters. Obsolated by the command dhparam
- **dsa**
  - management of DSA data
- **dsaparam**
  - management of DSA parameters
- **enc**
  - encryption/decryption

### Standard openssl commands (3)

- **errstr**
  - conversion of an error number to a string error
- **dhparam**
  - generation and management of Diffie-Hellman parameters
- **gendh**
  - generation of Diffie-Hellman parameters. obsolated by dhparam
- **gendsa**
  - generation of DSA parameters

### Standard openssl commands (4)

- **genrsa**
  - generation of RSA parameters (e.g. keys)
- **ocsp**
  - tool for Online Certificate Status Protocol
- **passwd**
  - generation of hashed passwords
- **pkcs12**
  - management of data PKCS#12
- **pkcs7**
  - management of data PKCS#7

### Standard openssl commands (5)

- **rand**
  - generation of pseudo-random numbers (bytes)
- **req**
  - management of requests of X.509 certificates - Certificate Signing Request (CSR)
- **rsa**
  - RSA data management
- **rsautl**
  - tool used to sign, verify, encrypt and decrypt data with RSA algorithm

### Standard openssl commands (6)

- **s_server**
  - a generic SSL/TLS server that accepts remote connections from clients that support the security protocol SSL/TLS. This application has been written to test the functionality of OpenSSL and has a minimal interface; internally however it uses almost all functionalities of OpenSSL.
- **s_client**
  - a generic SSL/TLS client that establishes an SSL/TLS connection with a remote SSL/TLS server; minimal interface too.
- **s_time**
  - SSL Connection Timer
- **sess_id**
  - management of session data of SSL/TLS connections

### Standard openssl commands (8)

- **smime**
  - processing of S/MIME messages
- **speed**
  - measure the performance of cryptographic algorithms
- **verify**
  - verifies/validates the X.509 digital certificates
- **version**
  - information on the version of the OpenSSL library
- **x509**
  - management of X.509 digital certificates
Preliminary steps: BASE64

- base64 encoding and decoding
  - Base 64 is an encoding/decoding system of binary data that uses 64 symbols.
  - commonly used when there is a need to encode binary data that needs to be stored and transferred over media that are designed to deal with textual data. This is to ensure that the data remains intact without modification during transport
  - is used mainly to encode/decode binary data in the e-mail messages to/from ASCII format.

BASE64 encoding algorithm

- splits the binary file in groups of 6 bits each, each of which contain consequently values from 0 to 63.
- each possible Base 64 value (Index) is converted in ASCII character, according to the Base 64 conversion table
- if the total number of bits is not a multiple of 6, there are inserted null bits (0) at the end, and in the encoding it is inserted the symbol '=' for each missing group of bits.

Conversion table: BASE64

<table>
<thead>
<tr>
<th>Base 64 Value → ASCII</th>
<th>M</th>
<th>a</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 → A</td>
<td>77</td>
<td>a</td>
<td>n</td>
</tr>
<tr>
<td>1 → B</td>
<td>97</td>
<td>b</td>
<td>n</td>
</tr>
<tr>
<td>3 → C</td>
<td>110</td>
<td>b</td>
<td>n</td>
</tr>
<tr>
<td>4 → D</td>
<td>110</td>
<td>c</td>
<td>n</td>
</tr>
<tr>
<td>5 → E</td>
<td>110</td>
<td>d</td>
<td>n</td>
</tr>
<tr>
<td>6 → F</td>
<td>110</td>
<td>e</td>
<td>n</td>
</tr>
</tbody>
</table>

BASE64 encoding algorithm

$ openssl base64 -in filename.bin -out filename.b64$

BASE 64

- to encode a file (data) in base 64 format:
  $ openssl base64 -in filename.bin -out filename.b64$

- to decode a file from the format base 64:
  $ openssl base64 -d -in filename.b64 -out filename.bin$

Exercise: encode/decode the text message "This is a message to encode" in/from base 64.

Symmetric algorithms

<table>
<thead>
<tr>
<th>name</th>
<th>chiave</th>
<th>blocco</th>
<th>note</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES</td>
<td>56 bit</td>
<td>64 bit</td>
<td>obsolete</td>
</tr>
<tr>
<td>3-DES</td>
<td>112 bit</td>
<td>64 bit</td>
<td></td>
</tr>
<tr>
<td>3-DES</td>
<td>168 bit</td>
<td>64 bit</td>
<td></td>
</tr>
<tr>
<td>IDEA</td>
<td>128 bit</td>
<td>64 bit</td>
<td></td>
</tr>
<tr>
<td>RC2</td>
<td>8-1024 bit</td>
<td>64 bit</td>
<td></td>
</tr>
<tr>
<td>RC4</td>
<td>variabile</td>
<td>stream</td>
<td></td>
</tr>
<tr>
<td>RC5</td>
<td>0-2048 bit</td>
<td>1-256 bit</td>
<td></td>
</tr>
<tr>
<td>AES</td>
<td>128-256 bit</td>
<td>128 bit</td>
<td>alias Rijndael</td>
</tr>
<tr>
<td>Blowfish</td>
<td>32-448 bit</td>
<td>64 bit</td>
<td></td>
</tr>
<tr>
<td>CAST-128</td>
<td>40-128 bit</td>
<td>64 bit</td>
<td></td>
</tr>
</tbody>
</table>
Symmetric crypto: exercises

- to encrypt/decrypt data with openssl tool you can use the openssl commands:
  - `enc`, `des3`, `bf`, `rc2`, `rc4`, ... ($ man enc)
- exercise: encrypt a file `ptext` with AES (128) in CBC mode, save the encrypted message in the file `ctext`:
  ```sh
  openssl enc -e -in ptext -out ctext -aes-128-cbc
  
  OpenSSL generates a symmetric key and an initialization vector (iv) from the password inserted by the user, for example:
  ```

```sh
$ openssl enc -e -in ptext -out ctext -aes-128-cbc -nosalt
$ openssl enc -e -in ptext -out ctext -aes-128-cbc -nosalt -p
```

- what do you observe?
  - `enc` can generate a key (and IV) from a password
  - the password must be recorded by the user, it provides weak security (is not random), insufficient as a key (e.g. subject to dictionary attacks)
  - `enc` calculates the hash of the password concatenated with a "salt"

Symmetric crypto: exercises (2)

- Exercise: execute again:
  ```sh
  $ openssl enc -e -in ptext -out ctext -aes-128-cbc -nosalt -p
  
  now execute (at least two times):
  ```

```sh
$ openssl enc -e -in ptext -out ctext -aes-128-cbc
```

- what do you observe?
  - `enc` can generate a key (and IV) from a password
  - the password must be recorded by the user, it provides weak security (is not random), insufficient as a key (e.g. subject to dictionary attacks)
  - `enc` calculates the hash of the password concatenated with a "salt"

Symmetric crypto: exercises (3)

- salt = random value and public (serves in decryption phase)
  - avoids known plaintext attacks
  - provides freshness: the same password, different keys at each encryption
- further details: function EVP_BytesToKey in crypto/evp/evp_key.c:
  ```
  for a detailed description of "salt" usage
  ```

```sh
$ openssl enc -d -in ctext -out dtext -K chiave -iv iv
```

- note the option --d (instead of --e), and that the name of files has changed

Symmetric crypto: exercises (4)

- What do the following commands?
  ```sh
  $ openssl enc -e -des3 -salt -in plaintext.txt -out ciphertext.bin
  $ openssl enc -e -des3-ede-ofb -d -in ciphertext.bin -out plaintext -pass pass:corsosicurezza
  $ openssl bf -cfb -salt -in plaintext -out ciphertext.bin
  $ openssl bf -cfb -salt -in plaintext -out ciphertext.bin -pass pass:diana
  $ openssl rc5 -in plaintext.txt -out ciphertext.bin -S C62CB1D49F158ADC -iv E9EDACA1BD7090C6 -K 89D4B167B604FAA3DBFFD030A3114B29
  ```

Hash algorithms

<table>
<thead>
<tr>
<th>name</th>
<th>block</th>
<th>digest</th>
<th>definition</th>
<th>note</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD28</td>
<td>128</td>
<td>16</td>
<td>RFC-1319 obsoleto</td>
<td></td>
</tr>
<tr>
<td>MD4</td>
<td>512</td>
<td>128</td>
<td>RFC-1320</td>
<td>obsolete</td>
</tr>
<tr>
<td>MD5</td>
<td>512</td>
<td>128</td>
<td>RFC-1321</td>
<td>good</td>
</tr>
<tr>
<td>RIPEMD</td>
<td>512</td>
<td>160</td>
<td>ISO/IEC 10118-3</td>
<td>optimum</td>
</tr>
<tr>
<td>SHA-1</td>
<td>512</td>
<td>160</td>
<td>FIPS 180-1</td>
<td>good</td>
</tr>
<tr>
<td>SHA-224</td>
<td>512</td>
<td>224</td>
<td>FIPS 180-2</td>
<td>optimum</td>
</tr>
<tr>
<td>SHA-256</td>
<td>512</td>
<td>256</td>
<td>FIPS 180-2</td>
<td>optimum</td>
</tr>
<tr>
<td>SHA-384</td>
<td>512</td>
<td>384</td>
<td>FIPS 180-2</td>
<td>optimum</td>
</tr>
<tr>
<td>SHA-512</td>
<td>512</td>
<td>512</td>
<td>FIPS 180-2</td>
<td>optimum</td>
</tr>
</tbody>
</table>
HASH: Commands + Exercises (1)

- to calculate a digest with openssl tool, you can use the openssl commands:
  - dgst, sha, sha1, md2, md4, md5, rmd160
- Exercise: calculate a digest with SHA1 algorithm on a file plaintext.txt (and print out on stdout the result in hexadecimal):
  - $ openssl sha1 plaintext.txt
  - $ openssl dgst –sha1 plaintext.txt
- calculate a digest with the algorithm SHA-256 and SHA-512 on a file plaintext.txt:
  - $ openssl dgst –sha256 plaintext.txt
  - $ openssl dgst –sha512 plaintext.txt

HASH: Exercises (2)

- Exercise: calculate a hash with SHA1 on the file plaintext.txt and save the hash in the file digest.txt:
  - $ openssl sha1 -out digest.txt plaintext.txt

Exercise: calculate a digest with SHA-256 and SHA-512 on a file plaintext.txt:

- $ openssl sha1 plaintext.txt
- $ openssl dgst –sha1 plaintext.txt
- $ openssl dgst –sha256 plaintext.txt
- $ openssl dgst –sha512 plaintext.txt

Asymmetric crypto: exercises (1)

- useful openssl commands:
  - genrsa, rsa, rsautil, gendsa, dsa, ..
- Exercise: generate a 2048 bit RSA key pair:
  - $ openssl genrsa –out rsa.key 2048
  - publicExponent: e = 2^16 + 1 (enc, verif)
  - privateExponent: d (dec, sign)
  - prime*: p, q (N = pq)
  - exponent*, coefficient: to speed up operations
  - standard PKCS1

RSA Encryption

- extract the public key from the file rsa.pubkey and view its content:
  - $ openssl rsa -in rsa.key -pubout -out rsa.pubkey
  - $ openssl rsa -pubin -in rsa.pubkey -text
- Exercise: create a message plain encrypt it with the RSA public key (2048-bit):
  - $ openssl rsautil -encrypt -in plain -pubin rsa.pubkey -out ctextrsa -pkcs
  - in practice, the maximum length of the message to encrypt is: N’=N – 11 (where N is the length in bytes of the RSA modulus). For example, for 2048-bit, N’=245. See PKCS #1 v2.1 (RFC 3447).
  - decrypt the message ctextrsa:
  - $ openssl rsautil -decrypt -in ctextrsa -inkey rsa.key

Asymmetric crypto: Examples (2)

- sign the content of the file plain with the RSA private key (rsa.key); the signature is saved in sig.bin:
  - $ openssl rsautil -sign -in plain -inkey rsa.key -out sig.bin
- verify the signature with the RSA public key (rsa.pubkey) and save the original data in the file prova_sig.txt:
  - $ openssl rsautil -verify -in sig.bin -pubin rsa.pubkey -out prova_sig.txt
Length of an RSA signature
- The length of the signature is proportional to the length of the key:
  - The digest has a fixed dimension
    - 16 bytes for MD5
    - 20 byte for SHA-1 and RIPEMD
  - The signature instead is long as the modulus (N) used:
    - 64 bytes for keys of 512 bits
    - 128 bytes for keys of 1024 bits
    - 256 byte for keys of 2048 bits

X.509 v3 Digital Certificates

Syntax of a certificate (ASN.1)
Certificate ::= SEQUENCE {
  tbsCertificate TBSCertificate,
  signatureAlgorithm AlgorithmIdentifier,
  signatureValue BIT STRING
}
TBSCertificate ::= SEQUENCE {
  version [0] Version DEFAULT v1,
  serialNumber CertificateSerialNumber,
  signature AlgorithmIdentifier,
  issuer Name,
  validity Validity,
  subject PublicKeyInfo
  subjectPublicKeyInfo SubjectPublicKeyInfo,
  issuerUniqueID [1] IMPLICIT UniqueIdentifier OPTIONAL
  -- if present, version must be v2 or v3
  subjectUniqueID [2] IMPLICIT UniqueIdentifier OPTIONAL
  -- if present, version must be v2 or v3
  extensions [3] Extensions OPTIONAL
  -- if present, version must be v3
}

Commands + Exercises (1)
- To create a demo CA, you need to execute the script:
  `$ /usr/ssl/misc/CA.pl -newca`
  
  ```
  demoCA
  key
  newcerts
  ca
  ```
- To request a new certificate:
  `$ openssl req -newkey rsa:2048 -pubkey -keyout usersakey.pem -out usercertreq.pem`
Digital certificates: Exercises (3)

- verify an X.509 certificate:
  ```bash
  $ openssl verify -CAfile demoCA/cacert.pem user_cert.pem
  ```

- revoke a certificate:
  ```bash
  $ openssl ca -revoke demoCA/newcerts/<serial>.pem
  ```

- issue the CRL that contains the revoked certificate:
  ```bash
  $ openssl ca -gencrl -out crl.pem
  ```

- view the content of a CRL:
  ```bash
  $ openssl crl -in crl.pem -text
  ```

- export a certificate:
  ```bash
  $ openssl pkcs12 -export -in user_cert.pem -inkey user_pkey.pem  
  -name "MY CERTIFICATE" -certfile demoCA/cacert.pem -out user_cert.p12
  ```

Digital certificates: Exercises (4)

- import the certificate in the browser:
  - In SeaMonkey: Edit->Preferences->Privacy&Security- 
  ->Manage Certificates->Import. Import the file user_cert.p12
  - In IE: Tools -> Internet Options -> Content -> Certificates -> 
  Import. Import user_cert.p12
  - You should view the demoCA among the trusted CA

- control the integrity of the certificate:
  - view the field SHA1 fingerprint of the certificate user_cert
  imported in the browser: serves to control the integrity of all the
  fields of the certificate (especially the public key !!)
  - In practice if the user wants to control the integrity of a certificate
    he could phone the CA and asks the value of the fingerprint of
    that certificate. If the fingerprint is the same, then this guarantees
    the integrity of the certificate (in all his parts)
  - the fingerprint is calculated as the hash of the certificate (in
    binary), like for example by executing:
    ```bash
    $ openssl sha1 user_cert.der
    ```