Cryptography with OpenSSL– Basic operations
Lab Report 2
Information Systems Security course (01TYM, 02KRQ)

prepared by:
Anuar Elio Magliari (s317033@studenti.polito.it)
George Florin Eftime (s303483@studenti.polito.it)
Alekos Interrante Bonadia (s319849@studenti.polito.it)

1. Symmetric cryptography

1. Symmetric cryptography exercises

- The encrypted message has been saved in the ctext.aes128 file. How long is the file ctext.aes128 with respect to the original plaintext file?
  - 48 bytes
- Now find out and write in the box the optional parameter that allows you to also show the used IV and the key
  - openssl enc -in ptext -e -p -out ctext.aes128 -aes-128-cbc -nosalt
- Has the key changed (even if have used the same password)? If yes, why? How long is the ciphertext file ctext.aes128 now? If the is different than in the previous encryption, explain what is the difference.
  1. The password changes every time we rerun the command due to the inclusion of the salt. The salt is concatenated with the password to generate a different key, even for the same passwords.
  2. The ciphertext file is 48 bytes in size because it consists of:
  3. IV (16 bytes)
  4. data to be encrypted (32 bytes)
  5. salt (8 bytes) + 8 bytes of padding (to reach 16 bytes, as a block size is 128/8 bytes).
- Which is the difference with respect to the previous command?
  - In this case, we applied a key derivation algorithm by inputting a specific password. We used PBKDF2 as the key derivation algorithm, and with the use of the SHA-512 encryption algorithm, it applies the same encryption 100,000 times. As for the file length, it has not changed; it remains at 48 bytes because, as mentioned earlier, it is composed of:
    - IV (16 bytes)

        - data to encrypt (32 bytes)
- Find out the corresponding command to use to decrypt the file ctext.aes128.pbkdf2.
    - openssl enc **-d** -aes-128-cbc -md sha512 -pbkdf2 -iter 100000 -nosalt -in ctext.aes128.pbkdf2 -p
- Now also find out the commands to encrypt the same data (ptext) with another symmetric algorithm (e.g. with 2 and 3 keys, in CBC mode). The encrypted message must be saved in a file named ctext.algorithm
    - openssl enc -in ptext -e -out ctext.3des -des-ede3-cbc -p -nosalt
    - len(key) = 24 byte => block size = 24/3 = 8 byte => (32 (data) + 8 (IV)) byte = 40 byte
- Find out the command to encrypt the plaintext with the AES algorithm in CTR mode.
    - openssl enc -e -in ptext -out ctext.aes128.ctr -aes-128-ctr -p -nosalt
- Compare now the length of the generated ctext.* file against the length of the original ptext. Is the size of the ciphertext file different from the size of the plaintext?
    - The size of the ciphered file generated is equal to the size of the plaintext (ptext) file due to how CTR mode works, where no padding is applied.
- Check the size of the file ctext.aes128.nopad and confront it with the size of the file ptext.
    - The size of the file generated is equal to 48 byte, because is composed by:
        - data encrypted (32 byte)
        - salt (8 byte) + 8 byte of padding (block’s size is 128/8 byte)
- Find out the commands to encrypt the same data (ptext) with the ChaCha20 algorithm and a key of your choice. Explain the difference between the encryption with the AES algorithm in the CBC and CTR modes and the encryption with the ChaCha20 algorithm.
    - openssl enc -e -in ptext -out ctext.aes128.cbc -aes-128-cbc -p -nosalt
    - openssl enc -e -in ptext -out ctext.aes128.ctr -aes-128-ctr -p -nosalt
    - openssl enc -e -in ptext -out ctext.chacha20 -chacha20 -p -nosalt

- In both cases (CBC and CTR), we are dealing with block ciphers with 128-bit keys, so each block is 16 bytes. However, they generate results of different sizes because it is related to how the different modes operate. CBC applies padding as part of its operation, while CTR does not apply padding and takes the leftmost bits. On the other hand, ChaCha20 is a stream cipher, so it generates a keystream equal in size to the file to be encrypted and then encrypts byte by byte with the data to be encrypted.

- Now find out the commands to decrypt the files ctext.algorithm. Find out the OpenSSL commands used to decrypt the cyphertext files and write them down.
  - openssl enc -d -in ctext.aes128.nopad -aes-128-cbc -nopad
  - openssl enc -d -in ctext.aes128.cbc -aes-128-cbc -p -nosalt
  - openssl enc -d -in ctext.aes128.ctr -aes-128-ctr -p -nosalt
  - openssl enc -d -in ctext.3des -des-ede3-cbc -p -nosalt
  - openssl enc -d -in ctext.chacha20 -chacha20 -p -nosalt
  - openssl enc -d -aes-128-cbc -md sha512 -pbkdf2 -iter 100000 -nosalt -in ctext.aes128.pbkdf2 -p

2. Brute force attack

Bob knows the original message was a text message. He wants to find out the key used by Alice to encrypt the message, by using ctext alice. After how many tries (at most) can he discover the Alice's secret key? Try out.

- If we assume that Alice has encrypted the file using the following command: "openssl enc -d -in ctext_alice -K 0xA -iv 0xE -chacha20", where the key and IV to be used for encrypting the file are specified, then Bob must try at most 16x16 combinations. This is because he needs to guess the corresponding key and IV to decrypt the file.

3. Performance evaluation

- encrypting

	100 B	10 kB	1 MB	100 MB
DES-EDE3	0,0000334 s	0,003345 s	0,0344 s	3,425 s
AES-128-CBC	0,0000007 s	0,000069 s	0,000703 s	0,0703 s
AES-192-CBC	0,0000009 s	0,000089 s	0,0009 s	0,0915 s
AES-256-CBC	0,0000009 s	0.000094 s	0,0009 s	0,0963 s

ChaCha20	0,0000005 s	0,000049 s	0,00490 s	0,4909 s

- decrypting

	100 B	10 kB	1 MB	100 MB
DES-EDE3	0,0000311 s	0,003311 s	0,0311 s	3,1851 s
AES-128-CBC	0,0000007	0,000068 s	0,000700 s	0,0700 s
AES-192-CBC	0,0000008 s	0,000083 s	0,000854 s	0,0854 s
AES-256-CBC	0,0000009 s	0,000094 s	0,000964 s	0,0965 s
ChaCha20	0,0000005 s	0,000049 s	0,00470 s	0,4940 s

2. Asymmetric cryptography

2.1 RSA Key generation

- Which of the parameters can be made public (are part of the public key) and which ones instead must be kept secret (are part of the RSA private key)?
  - Modulus
  - publicExponent
- Suppose you want to distribute your new RSA public key to your colleagues: write down the OpenSSL commands used to extract the RSA public key from the file rsa.key.name to the file rsa.pubkey.name, and to view its content:
  - openssl rsa -in rsa.key.name -pubout -out rsa.pub.name
  - openssl rsa -pubin -in rsa.pub.name -text

2.2 RSA encryption and decryption

- How can you use your RSA key pair to ensure confidentiality of the file plain? Which (RSA) key do you have to use?
  - We have to use the pub key
- Write down the OpenSSL command to encrypt the file plain, and save the result in the file encRSA (suggestion: the command pkeyutl).
  - openssl pkeyutl -encrypt -pubin -inkey rsa.pub.name -in msg.txt -out encRSA
- Which operation performs the following OpenSSL command and which key is used in this command?
  - This command encrypt the file plain with rsa private key.
  - openssl pkeyutl -encrypt -in plain -inkey rsa.key -out plain.enc.RSA.for.name

- Write down the command used to decrypt the message encrypted above:
  - openssl pkeyutl -decrypt -in plain.enc.RSA.for.name -inkey rsa.key.name -out decrypted_plain.txt
- Do you face any problem? Why (see the note below)?
  - This specific error often occurs in RSA encryption operations when the data to be encrypted exceeds the size limitations allowed by the RSA key's modulus. In RSA, the maximum amount of data that can be encrypted with a specific key is determined by the key's modulus size.

2.3 RSA signature generation and verification

- Which is the difference in terms of the RSA operations to be performed?
  - What will be change is which type of key is applied to encrypt the data, in this case to sign, will be used the private key to encrypt and the public key to decrypt.
- Which keys have you used for each of the above operations?
  - Private key for the signature, while public key for the confidentiality
  - openssl pkeyutl -verify -pubin -inkey rsa.key.alice.pub -in plain -sigfile plain.enc.RSA.for.alice.sign

2.4 EC Key generation

- Suppose you want to distribute your new EC public key to your colleagues: starting from the OpenSSL command ec, write down the command used to extract the EC public key from the file ec.key.name to the file ec.pubkey.name, and to view its content:
  - openssl ec -in ec.key.name -pubout -out ec.pubkey.name

2.5 EC signature generation and verification

- Which keys have you used for each of the above operations?
  - openssl pkeyutl -verify -pubin -in plain -inkey ec.pubkey.name -sigfile ecsig
    - Private key to encrypt the plain, so to sign the message
    - Public key to verify the signature

2.6 Performance evaluation

- openssl speed rsa*KeyLength* (e.g. openssl speed rsa15360)
- How does the performance changes with respect to the key length?

key length	sign	verify	sign/s	verify/s
2048	0.000558s	0.000017s	1792.8	58178.9
3072	0.001634s	0.000034s	612.0	29181.7
4096	0.003626s	0.000058s	275.8	17302.1
7680	0.032928s	0.000193s	30.4	5178.3
15360	0.166230s	0.000726s	6.0	1377.8

- What is the difference (in terms of performance) between operations requiring the use of the public key and the ones requiring the use of the private key?
  - The difference lies in the size of the public keys and private keys. Private keys are longer than public keys, so it takes more time to sign a message than to verify it because signing uses the private keys, while verification uses the public key.
- digital signature algorithms

algorithm	sign	verify	sign/s	verify/s
rsa1024	0.000083s	0.000005s	12080.0	190877.5
dsa1024	0.000101s	0.000068s	9905.0	14661.4
ecdsap160	0.0002s	0.0002s	5139.1	5151.1

ECDSA (Elliptic Curve Digital Signature Algorithm)

- What differences do you note?
  - RSA and DSA have quite similar perfomance
- Are the ECC operations more efficient than the RSA/DSA ones?
  - In terms of performance, the ECC operations are slower than RSA/DSA operations
- Is it efficient to sign or to verify with DSA/ECDSA?
  - No, it’s not so efficient as RSA
- Is it more efficient to sign or to verify with RSA?
  - In terms of performance is the best solution if we want to sign/verify fast
- digital signature algorithms (high security)

algorithm	sign	verify	sign/s	verify/s
rsa4096	0.003617s	0.000059s	276.5	17011.1
ecdsap256	0.0000s	0.0001s	45791.4	15617.7

- What differences do you note between the results obtained at this step and the ones obtained at the previous step?
    - RSA with 4096-bit keys is less efficient than 1024-bit keys. Meanwhile, ECDSA with 256 bits is more efficient than 160 bits.
- How does the performance decreases for RSA/DSA and for ECDSA with the increase of key length (and thus of the security strength)?

algorithm	**sign/s**	**verify/s**
rsa1024	12080.0	190877.5
rsa4096	276.5	17011.1
ratio	(12080 - 276) / 12080 = ~97%	(190877 - 17011) / 190877 = ~91%

algorithm	**sign/s**	**verify/s**
ecdsap160	5139.1	5151.1
ecdsap256	45791.4	15617.7
ratio	(5139 - 45791) / 5139 = -92%	(5151 - 15617) / 5151= -98%

In the case of ECDSA, the performance doesn’t decrease instead is increased.

3. Digest algorithms

3.1 Computation and verification of message digests

- Find out the correct OpenSSL commands to calculate the digests and write them down:
    - openssl dgst -sha*N* -out sha*N*.dgst msg
    - openssl dgst -sha3-*N* -out sha3-*N*.dgst msg
- What do you note when you compare the (above) two digests (are they similar, the same or different)?
    - openssl dgst -sha3-512 -out sha3-512.dgst msg
    - 8a393a0293323370a076040653af5b27d2271262a482195cba3e8dc36de449f7ec57a7c3ca3b6010d7ecbfa1996c378678674c46c97c5d8d5284ecbdce9ee154 (before the change)
    - 841e59a5fd87fa096536c7988dfc194f5b3277092dabba0b9df8a01d232c1f1d475c5d9eb526578cceec0b5abd8d3cddc5c905d39bdf39c5c85071c62a6fa66b (after the change)

- The message modified produces a different digest.

- Assume you have a digest composed of only the first hexadecimal digit in the file SHA256dgst. Are you able to find a collision? After how many tries?
  - Let D be the domain containing the words of interest, D' be the domain containing all possible digests (0-F), and given a digest function f that maps an element from D to an element in D', then if we have $A \in D$ such that A: $f(A) \rightarrow X \in D'$, then there exists a $B \in D$ such that B: $f(B) \rightarrow Y \in D'$ such that X == Y. This occurs at most on the sixteenth attempt.

3.2 Performance evaluation

	100B	10 KB	1 MB	100 MB
SHA256	0,0000002s	0,00002s	0,00023s	0,2373s
SHA512	0,0000002s	0,00002s	0,00017s	0,1758s

- How fast are the hash algorithms with respect to the symmetric encryption algorithms?
  - The hash algorithms are more performant than symmetric encryption algorithms

3.3 Application of digest algorithms: file integrity

- Write down the command used to calculate the digest of all files contained in the directory tree with SHA256. Save the digest in a file named hash list.
  - openssl dgst -sha3-512 -out hash_list tree/*
- How can an attacker change the content of some files so that its modification remains undetected (hint: the hash list is saved in a public, unprotected location.)?
  - The attacker can change what he wants in content of some files and after can recompute the digest of all files.
- What kind of protections can you adopt to defend from such attack? Enumerate at least two methods:
  - keyed digest, who creates the digest file can apply a key that is secret to all others (perhaps, the digest can be calculated putting the content between the key in order to avoid to attach other data to the original digest)
  - authenticate cryptography (e.g. AEAD)