# Lab Work 1

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

Last updated: Sep 12 13:31:23 2022

## Contents



# 1 Basics of OpenSSL

## 1.1 Installing OpenSSL

- Ubuntu: It can be easily installed with

  ```
  $ sudo apt-get install openssl
  ```

  but most likely it is already installed in your system. The previous line could install a quite outdated version (e.g., in Ubuntu 16.04, it installs OpenSSL 1.0.2g 01/03/2016, but it is now out of support!, but in Ubuntu 18.04 the installed OpenSSL version is 1.1.1). Use

  ```
  $ openssl version -a
  ```

  to obtain all the details of the installed version of OpenSSL. Newer (supported) versions can be installed from source code (current version is 3.0.5). However, the outdated version 1.0.2 is enough for education purposes.

- Windows: There are several ways to install OpenSSL in Windows systems. One of them is using CygWin to create a Linux-like environment and then install OpenSSL. On Windows 10 systems it is possible to enable its Linux subsystem, then install Linux on Windows. A Linux terminal can be launched from the Windows start menu.

(Generated by lineprocx v2.97)

- MacOS and iOS: Perhaps, using Linux on a virtual machine be the simplest option.
- Android: You can probably have OpenSSL installed in your Android phone. For instance, in a quite old phone running Android 5.1.1, with the Termux 0.83 App installed, it has OpenSSL 1.1.1d 10/09/2019, while in a quite modern Android 12 phone with the Termux 0.118 App installed from F-droid has OpenSSL 3.0.3 03/05/2022.

## 1.2 What is OpenSSL?

According to the documentation, OpenSSL is a cryptography toolkit implementing the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) network protocols and related cryptography standards required by them.

The openssl program is a command line tool for using the various cryptography functions of OpenSSL's crypto library from the shell. It can be used for

- Creation and management of private keys, public keys and parameters
- Public key cryptographic operations
- Creation of X.509 certificates, CSRs and CRLs
- Calculation of Message Digests
- Encryption and Decryption with Ciphers
- SSL/TLS Client and Server Tests
- Handling of S/MIME signed or encrypted mail
- Time Stamp requests, generation and verification

## 1.3 How to use the command-line tool

The command-line tool `openssl` provides access to the different tools implemented in the OpenSSL libraries.

You can obtain the list of openssl commands with

```
$ openssl help
```

Information about an individual openssl command (e.g. `enc`) and its arguments can be retrieved with

```
$ openssl enc -help
```

As usual, the detailed information can be found at the corresponding `man` page

```
$ man openssl
```

The different crypto algorithms implemented in your version of OpenSSL can be listed with

```
$ openssl list-cipher-algorithms
```

or with

```
$ openssl enc -ciphers
```

depending on the OpenSSL version.

### 1.4 How to compile your own OpenSSL code

The OpenSSL libraries (libcrypto and libssl) can be easily linked with the command line compiler options `-lcrypto` and `-lssl`.

For instance, you can write your C code and build an application with

```
$ gcc -o myapp myapp.c -lcrypto -lssl
```

## 2 Playing with stream ciphers in practice

### 2.1 Using the old RC4 stream cipher

RC4 is still implemented in OpenSSL 1.1.1, although its use is not recommended, but it has been deprecated in version 3.0. Thus, if you have OpenSSL 3.0 installed in your system, you cannot recreate the sample code given in this section. However, it is still worth reading because of the explanations about salted key generation procedures it contains.

Example of `openssl` call:

```
$ echo -n 'Hello World!'  | openssl enc -K '000102030405' -rc4 | xxd
```

where the 12-bytes long plaintext "Hello World!" is encrypted with RC4 (for a default key length of 128 bits) under the secret key `'000102030405'`, represented here in hexadecimal notation and padded with all-zero bytes to complete the required 128 bits. The result is a binary stream that is printed with `xxd`, producing the output

```
00000000:  a229 a949 2bf1 b974 0a37 dbbf            .).I+..t.7..
```

You can decrypt in a similar way with the option `-d`. In the previous example, you can pipe the output to a new `openssl` call to recover the original message:

```
$ echo -n 'Hello World!'  | openssl enc -K '000102030405' -rc4 | openssl enc -d -K '000102030405'
-rc4 | xxd
```

producing the output

```
00000000:  4865 6c6c 6f20 576f 726c 6421            Hello World!
```

Using the same key to encrypt several messages is completely insecure, but there is no standard way to introduce some randomness or variability to produce different keystreams under the same secret key. In some old systems a initialization value IV was prepended to the secret key to form the actual RC4 key. For instance a 3-byte IV `01FF03` can be manually prepended to the previous key:

```
$ echo -n 'Hello World!'  | openssl enc -K '01FF03000102030405' -rc4 | xxd
```

and every new plaintext is encrypted with a different IV. Now, you need the IV value to be able to decrypt

the ciphertext. Encrypting instead the all-zero message directly gives you the keystream.

Prepending or appending the IV to the key is insecure in RC4, but there are other options to derive a different RC4 key in every encryption session. For instance, you can use a more involved Key-Derivarion Function to transform a secret key (called 'passphrase') and a varying public value (called 'salt') into a (probable) unique key. In OpenSSL you can use this mechanism to encrypt a message. For instance with the passphrase "Not very secure" and the salt `00010203` (in hexadecimal) you can generate an RC4 key and encrypt the same plaintext used before:

```
$ echo -n 'Hello World!'  | openssl enc -k 'Not very secure' -S '00010203' -p -rc4 | xxd
```

This line produces something like

```
00000000:  5361 6c74 6564 5f5f 0001 0203 0000 0000  Salted__........
00000010:  7361 6c74 3d30 3030 3130 3230 3330 3030  salt=00010203000
00000020:  3030 3030 300a 6b65 793d 4545 4633 4642  00000.key=EEF3FB
00000030:  3632 3235 3141 3238 4438 4633 3342 3334  62251A28D8F33B34
00000040:  3337 3445 3741 3636 4431 0a5f 6627 1ad0  374E7A66D1._f'..
00000050:  e74b 3f8c 2bc2 78                        .K?.+.x
```

containing information about the salt used and the RC4 key computed from the passphrase and the salt, as well as giving the resulting ciphertext. The `-p` option produces the salt and key information (the salt is padded with zeroes up to the necessary length according to the Key Derivation Function used by `openssl`). The above output corresponds to `openssl` version 1.0.2g, while in version 1.1.1d the output is

```
00000000:  5361 6c74 6564 5f5f 0001 0203 0000 0000  Salted__........
00000010:  7361 6c74 3d30 3030 3130 3230 3330 3030  salt=00010203000
00000020:  3030 3030 300a 6b65 793d 4136 4339 4444  00000.key=A6C9DD
00000030:  3545 4336 3331 3833 4632 3946 3234 3032  5EC63183F29F2402
00000040:  4145 4437 3241 3738 4543 0a7c 3110 7543  AED72A78EC.|1.uC
00000050:  4611 c53c 40a7 4e                        F..<@.N
```

Without the `-p` option, the output is only the ciphertext, which includes the 16-bytes long header describing the salt used to derive the key. Using version 1.0.2g, the ciphertext is

```
00000000:  5361 6c74 6564 5f5f 0001 0203 0000 0000  Salted__........
00000010:  5f66 271a d0e7 4b3f 8c2b c278            _f'...K?.+.x
```

Observe that the above outputs generated by different OpenSSL versions reflect different symmetric keys for the same passphrase. This is due to the different Key Derivation Functions (KDF) between OpenSSL versions: KDF used in old versions like 1.0.2g is based on the deprecated MD5 hash function, while it have been updated to the more secure SHA256 hash function in version 1.1.1. Namely, the derived symmetric key in older versions of OpenSSL is just computed as the (truncated) MD5 output of the concatenation of the passphrase and the salt value. Indeed, the output of

```
md5sum <(echo -n 'Not very secure' ; echo -n '0001020300000000' | xxd -p -r)
```

is exactly the key generated above by OpenSSL version 1.0.2g, while the key generated by version 1.1.1d is obtained by replacing `md5sum` by `sha256sum`, and truncating the output.

Nowadays, from 2017 on, a more secure iterative construction of a KDF, named PBKDF2, is recommended in order to protect passphrases from dictionary attacks. From OpenSSL version 1.1.1 on this KDF can be enabled with the option `-pbkdf2`, and the number of iterations can be set with `-iter`. A number of iterations above 10000 is currently recommended. In addition, a good choice for the salt is a random 128-bit value.

### 2.2 Using the chacha20 stream cipher

The newer `chacha20` is a modern stream cipher with a reasonable keysize (256 bits) and a precise way to use an initialization value (64 bits). An iterative mixing algorithm transforms a key, an initialization value and a counter (64 bits) into a keystream block (512 bits). As usual, the keystream blocks are XORed with the message blocks to produce the cipherstream.

The message, key and cipher streams are not byte-oriented, but they use a block size of 512 bits.

The key stream blocks can be generated independently of the others (random-access). In other stream ciphers (like RC4) the keystream has to be computed sequentially.

Chacha is not implemented in earlier versions of OpenSSL, like 1.0.2g, but it is in version 1.1.1d.

```
$ echo -n 'Hello World!'  | openssl enc -K '00' -iv '00' -p -chacha20 -nosalt | xxd
```

produces

```
00000000:    6b65 793d 3030 3030 3030 3030 3030 3030  key=0000000000000
00000010:    3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
00000020:    3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
00000030:    3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
00000040:    3030 3030 0a69 7620 3d30 3030 3030 3030  0000.iv=00000000
00000050:    3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
00000060:    3030 3030 3030 3030 300a 3edd 8cc1 cfd1  000000000.>.....
00000070:    6aff 3231 0ec4                           j.21..
```

where, the key and the iv values where padded with zero-bytes to the appropriate lengths.

A (slightly) more realistic call would be

```
$ echo -n 'Hello World!'  | openssl enc -k 'Not very secure' -S '00010203' -chacha20 | xxd
```

which uses a passphrase and a value for the salt. The output is now

```
00000000:    5361 6c74 6564 5f5f 0001 0203 0000 0000  Salted__........
00000010:    5593 58dc 45b5 0fd5 25ab 1e57            U.X.E...%..W
```

## 3 A (somewhat) practical attack against RC4

### 3.1 The attack scenario

In this section we propose a easy-to-implement key-recovery attack against a particular mode of operation of RC4. We need several assumptions to launch a successful attack:

- The initialization value `iv` is a 3-byte counter prepended to a 13-byte long-term key, making a 16-byte string that is used as the actual RC4 key.

- The `iv` is incremented in every new encryption.
- RC4 is used in a communication system to send some well-structured packets with a constant header of at least one byte `m[0]`.
- The attacker has access to many encryptions, so that he can wait for the use of some specific `iv` values.

The previous assumptions can be fulfiled in some systems implementing handshake packets encrypted with RC4.

The attack is based on the following facts:

1. For any byte `x`, using `iv=01 FF x` results in a keystream such that its first byte equals `x+2` with high probability.
2. The first keystream byte produced with `iv=03 FF x` is, with a noticeable probability, `x+6+k[0]`, where `k[0]` denotes the first byte of the long-term key. Similarly, `iv=04 FF x` produces the first keystream byte equal to `x+10+k[0]+k[1]` with a noticeable probability.
3. In general, for $i$ ranging from 0 to 12, using `iv=z FF x` where `z` is the hexadecimal representation of $i+3$ often produces the first keystream byte equal to `x+d[i]+k[0]+k[1]+...+k[i]`, where `d[i]` is the constant $1+2+\ldots+(i+3)$.

## 3.2 Simulating the attack

In order to simulate the attack, you can first simulate the channel eavesdropping. To do that, start generating a 13-byte random key and a one-byte random message `m[0]` (the first byte of the message suffices to perform the attack, assuming that it is a value fixed by the communication protocol, like a packet length, version number, etc.). Then, for each of the values of `iv` mentioned before, concatenate it with the key and encrypt `m[0]` with the RC4 implementation in OpenSSL. If you have OpenSSL version 3.0, you will need a custom implementation of RC4, like the provided `rc4enc` tool.

Then collect all one-byte ciphertexts `c[0]` and store them along with the corresponding values of the `iv` into a table or file, following the structure

```
0X01FF00 0XDB
0X01FF01 0X60
0X01FF02 0XE8
0X01FF03 0X74
...
```

You can use separate files for every collection of values (one file for the 256 values `iv=01FFx`, another file for `iv=03FFx`, etc.).

Now you proceed with the actual attack (you now pretend that you do not know the key or the message and you try to learn both from the gathered data).

Using fact 1 described above, compute the 256 values of `c[0] XOR (x+2)` and choose the most frequent one as your guess for `m[0]`. Notice that all operations are performed with unsigned bytes, so you must use 8-bit arithmetic (or just take modulo 256 to the result of every computation, so that, for instance, `0XFF+2=0X01`).

Tools like `sort` or `unique` can be useful to detect the most repeated value. You can also use standard tools (spreadsheets) of a script in your preferred programming language (perl, python, C) to process the collected ciphertext bytes to find the most repeated values and end the simulated attack.

In a similar way, use fact 2 to recover `k[0]` as the most frequent value of `(c[0] XOR m[0])-x-6`, where `m[0]` is the guessed value obtained before.

Compute also the most frequent value of `(c[0] XOR m[0])-x-10-k[0]`, where again `m[0]` and `k[0]` are the previously guessed values. Remember that each step uses a different collection of values of `iv`.

You can try to use fact 3 to guess all the remaining bytes of the key.

### 3.3 Practical work

- (Paper and pencil) From the description of the key scheduling phase of RC4, show facts 1 and 2 described above.
- Implement the attack simulation described above, first choosing a random key and one-byte message, then generating the files containing all the gathered ciphertexts, and finally guessing the secret values from this data.
- Observe that sometimes the guesses are wrong. Compare the guessed values to the original ones. An example of the trace of a possible attack is the following:

```
key is 7e1a0bbc8c770667be44dce10c and message is 90
Gathering keystream first bytes for IV=01FFxx ...  done
Guessing m[0] ...  done
   Guessed m[0]=90 (with freq.    31)             *** OK ***
Gathering keystream first bytes for IV=03FFxx ...  done
Guessing k[0] ...  done
   Guessed k[0]=7e (with freq.    16)             *** OK ***
Gathering keystream first bytes for IV=04FFxx ...  done
Guessing k[1] ...  done
   Guessed k[1]=1a (with freq.    20)             *** OK ***
Gathering keystream first bytes for IV=05FFxx ...  done
Guessing k[2] ...  done
   Guessed k[2]=0b (with freq.    17)             *** OK ***
Gathering keystream first bytes for IV=06FFxx ...  done
Guessing k[3] ...  done
   Guessed k[3]=bc (with freq.    13)             *** OK ***
Gathering keystream first bytes for IV=07FFxx ...  done
Guessing k[4] ...  done
   Guessed k[4]=8c (with freq.    19)             *** OK ***
Gathering keystream first bytes for IV=08FFxx ...  done
Guessing k[5] ...  done
   Guessed k[5]=77 (with freq.    10)             *** OK ***
Gathering keystream first bytes for IV=09FFxx ...  done
Guessing k[6] ...  done
   Guessed k[6]=06 (with freq.    12)             *** OK ***
Gathering keystream first bytes for IV=0AFFxx ...  done
Guessing k[7] ...  done
   Guessed k[7]=67 (with freq.     7)              *** OK ***
Gathering keystream first bytes for IV=0BFFxx ...  done
Guessing k[8] ...  done
   Guessed k[8]=be (with freq.    12)             *** OK ***
Gathering keystream first bytes for IV=0CFFxx ...  done
Guessing k[9] ...  done
   Guessed k[9]=44 (with freq.    13)             *** OK ***
Gathering keystream first bytes for IV=0DFFxx ...  done
Guessing k[10] ...  done
   Guessed k[10]=dc (with freq.     7)              *** OK ***
Gathering keystream first bytes for IV=0EFFxx ...  done
Guessing k[11] ...  done
   Guessed k[11]=e1 (with freq.    13)              *** OK ***
```

```
Gathering keystream first bytes for IV=0FFFxx ...  done
Guessing k[12] ...  done
   Guessed k[12]=0c (with freq.    10)             *** OK ***
```

You have some text files available, files `bytes_01FFxx.dat, ..., bytes_0FFFxx.dat`, generated with an unknown random key and plaintext byte. Try to apply the attack to those files and retrieve the first bytes of the key and the ciphertext byte.

You are expected to hand in a short report containing:

- The solutions to the paper and pencil parts.
- A short explanation of how your scripts implementing the attack are organized.
- Some execution traces.
- Your guess of the message and key used to generate the provided text files mentioned above.

You must also hand in the scripts implementing the attack.

---