Cybersecurity MSc: Applied Cryptography 2023-24

Practical Exercises: Symmetric Cryptography

September 14, 2023 Due date: no date

Changelog

- v1.0 - Initial version.

Introduction

In order to elaborate this laboratory guide it is required to install the Java Development Environment (JDK), Python 3 or the C compiler and development environment.

Because you will need to visualize binary files, the `ghex` application may also be useful. It is available for installation in the Linux repositories using `apt-get install ghex`.

The examples provided will use both Java, Python and C. For Python you need to install the `cryptography` module. For C we will use the EVP (Digital Envelope) functions from the Crypto library that is part of OpenSSL, and for that you need to use install the packages `libssl1.0-dev` and `libssl1.1`.

1 Elements of interest for each language

1.1 Java

- javax.crypto.**Cipher** An instance of the `Cipher` allows to encrypt original contents or decrypt cryptograms. Some important methods are:
  - `getInstance`: creates an instance of a encryption/decryption engine for a given algorithm;
  - `init`: initializes the engine instance; typically used to set the key to be used, at least;
  - `update`: performs an incremental encryption/decryption with the engine instance. Engines that work sequentially can keep some internal state across successive calls;
  - `doFinal`: terminates a sequential process (not required for non-sequential processes).
- javax.crypto.**KeyGenerator** An instance of the class `KeyGenerator` is a generator of symmetric keys. Some important methods are:
  - `getInstance`: creates an instance of a generator for producing keys for a given cipher algorithm;
  - `init`: initializes the key generator with some elements for helping the key production (key size, when it can vary for a given algorithm, such as AES, randomness source, etc.);
  - `generateKey` effectively creates a symmetric key.
- javax.crypto.**SecretKey** An instance of a class implementing the interface `SecretKey` contains a symmetric key for a given cipher algorithm.

1.2 Python

- The cryptography module is both a frontend, high-level interface for dealing with cryptography and a default backend implementation of the fundamental cryptographic primitives.
- The Python interpreter distinguishes bytes from text characters, and thus byte arrays from strings. A string is more than a byte array, it has also an encoding (e.g. UTF-8, Unicode, etc.). You can get the bytes of characters (strings) using the `bytes` function, and you can convert bytes to characters (strings) by using the method `decode` on a byte array (with a target encoding).

  By default, in cryptography we always use bits and bytes, nothing else. So, do not attempt to solve interpreter errors by changing everything to text (characters and strings)!

1.3 C

- Functions `EVP_···_ex` are functions that can be parameterized with an implementation engine, whilst functions without the `_ex` suffix use the default library engine.
- Functions `EVP_CipherInit`, `EVP_CipherUpdate` and `EVP_CipherFinal` form the high-level interface for handling (sequential) encryption and decryption operations.
- Functions `EVP_Encrypt···` are the high-level interface for handling encryptions, whilst `EVP_Decrypt···` handle decryptions (using the previous ones).
- Functions `EVP_CIPHER_CTX_set···` help to parameterize a cipher context, possibly with items not available in the `EVP_···Init` functions.
- A structure `EVP_CIPHER_CTX` describes the current context of a cryptographic transformation, and it is created with `EVP_CIPHER_CTX_new` and released with `EVP_CIPHER_CTX_free`. This context is used to store fundamental data elements (e.g. key and IV) used to process cryptographic transformations and to buffer input data across sequential operations for dealing with block alignment and padding.

- A structure `EVP_CIPHER` describes generic attributes of a given cryptographic transformation. Usually it does not need to be filled manually, one can use predefined functions for that (e.g. `EVP_aes_256_cbc` returns a pointer to such a structure describing an AES cipher with a 256-bit key operating in CBC mode).

2 Creation of a symmetric key from a password

Develop a program to generate a symmetric key for the `AES` algorithm, and save it to a file. The key should be generated from a password, in order to implement a process known as Password-Based Encryption (PBE). The password should be provided as the first argument of the program and the file name as the second.

Tip: you can use less arguments and consider the use of `stdout` in the absence of file specifications.

The program should generate 128-bit keys, as this is required by the AES algorithm. For dealing with PBE, we will use a password-to-key transformation function known as PBKDF2 (Password-Based Key Derivation Function 2). This is a generic, multi-iteration hashing function, which can be parametrized with another keyed hashing function; we will use HMAC with SHA-1.

Tip: you can use less arguments and consider the use of `stdout` in the absence of file specifications.

Tip: the output should be language-independent, thus you should get exactly the same output using different programming languages.

Note: functions such as PBKDF2 are often used to compute password transformations used in authentication processes. The final goal is to avoid storing passwords in cleartext for matching with the ones providing by people. However, for preventing exhaustive search attacks looking for a password suitable for a given stored transformation, such transformations are often randomized with an element called **salt**. A salt does not need to be secret, it only needs to be randomly assigned once for a given password and used to transform it thereafter; that way, we can break massive password guessing attacks, such as the one one can implement with a rainbow tables, because it is infeasible for them to deal with different salt values.

The following code samples exemplify how to convert a textual password, stored in the string `pwd`, into a byte array (`key`) with PBKDF2 (with HMAC and SHA-1) using Java, Python and C. Note that when non-ASCII characters are part of the password, the output of all programs will only be equal iff the encoding used in Java and C is also UTF-8.

```
import javax.crypto.spec.PBEKeySpec;
import javax.crypto.spec.SecretKeySpec;
import javax.crypto.SecretKeyFactory;
import javax.crypto.SecretKey;

// PBKDF2 specifications: password, salt, iterations and output size
//
// salt: a byte array intended to produce different outputs for the same password
// iterations: the number of times the PBKDF2 iterates internally over a generator function
// output size: the number of bits we want to generate from the password (128 for AES)

byte[] salt = new byte[1];
salt[0] = 0; // We need to provide one salt ...

// First, we create a specifications object with all the PBKDF2 parameters we want

PBEKeySpec pbeKeySpec = new PBEKeySpec( pwd.toCharArray(), salt, 1000, 128 );

// Then, we create key material (a byte array) using PBKDF2 + HMAC(SHA-1)

SecretKeyFactory skf = SecretKeyFactory.getInstance( "PBKDF2WithHmacSHA1" );
byte[] key = skf.generateSecret( pbeKeySpec ).getEncoded();

// Write key in a file
```

The following code does the same in Python:

```
import sys
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.backends import default_backend

# The PBKDF2 generator of Python receives as input the number of bytes to generate,
# instead of bits

salt = b'\x00'
kdf = PBKDF2HMAC( hashes.SHA1(), 16, salt, 1000, default_backend() )
key = kdf.derive( bytes( pwd, 'UTF-8' ) )

# Write key in a file
```

The following code does the same in C:

```
#include <stdint.h>
#include <openssl/crypto.h>
#include <openssl/evp.h>

#define KEY_LEN 16 // bytes

char salt = 0;
uint8_t key[KEY_LEN];

// The PBKDF2 generator of OpenSSL receives as input the number of bytes to generate,
// instead of bits

PKCS5_PBKDF2_HMAC_SHA1( pwd, -1, &salt, 1, 1000, sizeof(key), key );

// Write key in a file
```

3 Ciphering a file using the `AES` algorithm

Write a program to encrypt the contents of a file using the `AES` cipher and a key previously generated and stored in a file. The program should accept three parameters: the key file, the input and the output files.

Tip: you can use less arguments and consider the use of `stdin` and `stdout` in the absence of file specifications.

The following code samples in Java, Python and C exemplify how to encrypt a file's contents with AES in CBC mode with a PKCS \$7 padding. The encryption key is loaded from a key file and the IV is randomly generated and stored in the beginning of the file with the cryptogram.

Tip: the output should be language-independent, thus you should get exactly the same output using different programming languages. However, this is only true if you use exactly the same parameters. In the example provided this does not happens because they use a random IV, but would happen for a fixed IV.

Check the size of the resulting encrypted files for each input file. Explain why they are always bigger and the actual size increment.

Create a new program performing a similar encryption but without padding and observe the result. You may find that you are constrained in the size of the files that you are able to cipher. In other words, your program will not work properly with input files with a size that is not a multiple of 16 bytes.

```
import javax.crypto.spec.SecretKeySpec;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.SecretKey;
import java.security.SecureRandom;
import javax.crypto.Cipher;

// Read key bytes from key file (into variable keyBytes)
...

// Setup AES key with the key to encrypt

SecretKey key = new SecretKeySpec( keyBytes, "AES" );

// Create a cipher engine given the algorithm (AES), the encryption mode (CBC)
// and the padding (PKCS \#5)

Cipher c = Cipher.getInstance( "AES/CBC/PKCS5Padding" );

// Set the IV parameters (required for CBC) to a random value.
// The IV must have a size equal to the cipher's block size

byte[] iv = new byte[c.getBlockSize()];
SecureRandom random = new SecureRandom();
random.nextBytes( iv );

// Set the cipher engine to encrypt with the intended key and IV

c.init( Cipher.ENCRYPT_MODE, key, new IvParameterSpec( iv ) );

// Open input file for reading and output file for writing
...

// Write the IV in the output file
...

while (...) { // Cicle to repeat while there is data left on the input file
    // Read a chunk of the input file to the plaintext variable
    ...

    // The length of the plaintext data should be stored in pLen

    ciphertext = c.update( plaintext, 0, pLen );

    // Store the ciphertext in the output file
    ...
}

// Perform the encryption of the last plaintext contents + the padding

ciphertext = c.doFinal();

// Store the ciphertext in the output file
...
```

The following code does the same in Python:

```
import os
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.backends import default_backend

# Read key bytes from key file (into variable key)
...

# Setup cipher: AES in CBC mode, w/ a random IV and PKCS #7 padding (similar to PKCS #5)

iv = os.urandom( algorithms.AES.block_size // 8 );
cipher = Cipher( algorithms.AES( key ), modes.CBC( iv ), default_backend() )
encryptor = cipher.encryptor()
padder = padding.PKCS7( algorithms.AES.block_size ).padder()

# Open input file for reading and output file for writing
...

# Write the contents of iv in the output file
...

while True: # Cicle to repeat while there is data left on the input file

    # Read a chunk of the input file to the plaintext variable
    ...

    if not plaintext:
        ciphertext = encryptor.update( padder.finalize() )

        # Write the contents of ciphertext in the output file

        break
    else:
        ciphertext = encryptor.update( padder.update( plaintext ) )

        # Write the ciphertext in the output file
```

The following code does the same in C:

```c
#include <stdint.h>
#include <openssl/crypto.h>
#include <openssl/evp.h>
#include <openssl/aes.h>
#include <openssl/rand.h>

#define KEY_LEN 16 // bytes

EVP_CIPHER_CTX * ctx;
uint8_t key[KEY_LEN];
uint8_t iv[AES_BLOCK_SIZE];

// Read key bytes from key file (into variable key)
...

// Setup cipher: AES in CBC mode, w/ a random IV and PKCS #7 padding (similar to PKCS #5)

RAND_bytes( iv, sizeof(iv) );
ctx = EVP_CIPHER_CTX_new();
EVP_CipherInit( ctx, EVP_aes_128_cbc(), key, iv, 1 );

// PKCS #7 padding is on by default; use the following call to disable it
// EVP_set_padding( ctx, 0 );

// Open input file for reading and output file for writing
...

// Write the contents of iv in the output file
...

while (...) {
    uint8_t ciphertext[ /* allocate as much bytes as for variable plaintext (>=
        AES_BLOCK_SIZE) */ ];
    int cLen;

    // Read a chunk of the input file to the plaintext variable
    ...

    // The length of the plaintext data should be stored in pLen

    EVP_CipherUpdate( ctx, ciphertext, &cLen, plaintext, pLen );

    // Write the first cLen bytes of ciphertext in the output file
    ...
}

EVP_CipherFinal( ctx, ciphertext, &cLen );
// Write the first cLen bytes of ciphertext in the output file
...

EVP_CIPHER_CTX_free( ctx );
```

4 Deciphering a file using AES

Develop a program similar to the previous but able to decipher a file. Three arguments should be sent to the program: the key file, file to decipher and the output file.

Tip: Try to use a program written in a different language than the one with which you did the encryption.

5 Ciphering and deciphering a file using a given algorithm

Modify the previous programs in order to accept an additional parameter stating the algorithm to use. Consider that two options can be provided: `AES` and `DES`. The change should be minimal.

6 Cipher modes

6.1 Initialization Vector

Some cipher modes requiring feedback information (`CBC`, `OFB`, `CFB` and `CTR`) must use an Initialization Vector (IV).

Modify the previous programs to accept as argument a file name, which should contain a definition of the cipher algorithm, the cipher mode and an IV (assume a default padding when required). The content of this file should be created by the ciphering application and used by the decryption application to initialize the decryption engine. The IV in that file should be a random value generated when ciphering.

Note: you do not need to store the IV anymore in the beginning of the encrypted file.

Take in consideration that only the `ECB` cipher mode does not require the use of an IV.

6.2 Propagation of patterns

In this exercise the goal is to analyze the impact of using `ECB` and `CBC` from the perspective of the propagation of patterns between the plaintext and the corresponding cryptogram.

The approach followed will be to use a BMP file, cipher it, and visualize the resulting cryptogram. The BMP format is very simple and as long as the first 54 bytes (the header) are kept unchanged, the remaining content will be shown as an image.

With the program you developed, and using the `ECB` mode, with any encryption algorithm, cipher the file `security.bmp` to another file, named `security-ecb.bmp`. Then restore the BMP header of the ciphered file with the original one. This will allow for any graphics application to interpret the file as a BMP file, even if the picture data is ciphered.

The following command can be used to fix the header from the encrypted contents to the original ones:

```
dd if=security.bmp of=security-ecb.bmp bs=1 count=54 conv=notrunc
```

Open both files with any BMP viewer, and compare the results.

Repeat the same operation with the same algorithm and over the same file `security.bmp`, but now using the `CBC` mode. You should produce a file named `security-cbc.bmp`. Restore the header, view both images and compare the result.

Repeat the above steps for other algorithms and cipher modes. What can you conclude?

6.3 Error propagation

In this exercise we will analyze the impact of errors in the ciphertext. That is, the effect of modifications to one or more bits in the ciphertext, and then deciphering the ciphertext into the clear text, when using `ECB`, `CBC`, `OFB` and `CFB`.

Using the program developed, with any cipher algorithm, and the `ECB` cipher mode, cipher the image that was provided with this assignment. Do not restore the header!

Using an hex editor, such as `ghex`, select a random byte and take notice of this byte. Then flip one bit to the opposite value. As an example, you can use address `0xec00` which encodes the lower right pixel of the dot in the exclamation mark, after the word RSA.

Decipher the file you just modified using the same algorithm and mode. View both the original image, and the one you just obtained. Then compare their content using an hex editor. In particular, focus in the byte that you just changed, and the surrounding bytes. You can also use the `cmp -bl firstFile secondFile` command.

Repeat these steps with the remaining cipher modes, and for each find what is the impact of errors in the ciphertext. Also, define which cipher modes are more and less sensitive to errors in the ciphertext (considering the amount of errors in the final image).

7 Triple DES

The `DES` algorithm uses keys with 56 bits, and was considered insecure some time after its creation. However, it was created a method to increase its security by doubling or tripling the key size. This method is frequently called `TripleDES` or `3DESede`. What this method introduces is the notion of multiple operations over the text, using different keys and is a good example of a cipher reinforcement method.

When using two keys (112 bits), `TripleDES` is implemented by calculating:

$$E_{k1}\left(D_{k2}\left(E_{k1}\left(text\right)\right)\right)$$

When using three keys (168 bits), `TripleDES` is implemented by calculating:

$$E_{k3}\left(D_{k2}\left(E_{k1}\left(text\right)\right)\right)$$

Where k_i is a key, D is a decipher operation, and E is a cipher operation. This method is based on the fact that deciphering a cryptogram with a wrong key is equivalent to cipher it again.

Implement a program which applies this method to the `DES` cipher. Please take in consideration that, when ciphering, only the first cipher operation should use padding! When deciphering, the last operation should use padding[1].

8 Cipher performance

An important aspect of the different ciphers is their performance in common hardware, which varies by a great amount. Taking in consideration the ciphers available in Java (`Blowfish`, `AES`, `DES`, `RC2`, `RC4`, `ARCFOUR`, and `3DESede`) implement a program to benchmark each cipher. Consider blocks with size ranging from 16 bytes to 8192 bytes. In order to run the benchmark, consider the method `System.currentTimeMillis()` and see how many time it takes to do $10,000,000$ cipher operations.

Tip: do not consider for timing the first cipher/decipher operation, as it will "warm" data and code caches.

Tip: For more accurate timming, consider performing some loop unrolling (repeat the same instruction several times in the loop body). This is more relevant for accurately evaluate fast ciphers than slow ones.

[1] Java, Python and C support `3DESede` natively, but you should not use it unless you wish to test your implementation.

References

- The Java Tutorial: Security Features in Java SE
  `https://docs.oracle.com/javase/tutorial/security/index.html`
- Java Documentation: Security
  `http://docs.oracle.com/javase/8/docs/technotes/guides/security/index.html`
- PKCS #5: Password-Based Cryptography Specification Version 2.0
  (`https://tools.ietf.org/html/rfc2898`)
- Welcome to pyca/cryptography
  (`https://cryptography.io`)
- Libcrypto API
  (`https://wiki.openssl.org/index.php/Libcrypto_API`)