Computer Science Department
University of Crete

CS458: Introduction to Cryptography
Assignment 2 – Symmetric Cryptography

Deadline: ~~28/11~~ 2/12 23:59

Introduction

The main purpose of this assignment is to offer you the opportunity to get familiar with the AES algorithm, modes supporting authenticated encryption and symmetric encryption in general.

Setup

When working on this assignment, remember to activate your Python virtual environment, that you were instructed to install in the first assignment, if you don't want Python modules to be installed system-wide:

```
cd hy458
source env/bin/activate
```

For this assignment you will have to use the "cryptography" Python library:

```
pip install cryptography
```

Follow the libraries documentation for implementing the encryption and decryption processes:
https://cryptography.io/en/latest/hazmat/primitives/symmetric-encryption/

Also, you can use the "argparse" Python module to parse any necessary command line arguments.

Any data used inside the program should be represented as bytes. If you need to handle ciphertext or other input/output, you can convert it to the hexadecimal representation of the bytes. **You have to handle any conversions yourself!**

Also, your code should be able to be integrated into any system without changes. Use functions with appropriate parameters and minimize top-level code. For example, you can create a main function that is called as:

```
import sys

def f():
    print(“Hello world”)

def main(args):
    print(args)
    f()

if __name__ == “__main__”:
    main(sys.argv[1:])
```

As you can see very little code runs on the top-level. Everything is wrapped in functions making it easy to integrate it in test files or larger systems.

1. CBC: Basic functionality [10%]

- Create a Python function that, given the plaintext, key, and initialization vector (as bytes), will encrypt them using AES-128-CBC. The function should return the ciphertext generated by the algorithm. Your job is to implement AES-128-CBC encryption using AES-128-ECB as a base. Look for the relevant lecture slides to understand how this mode is implemented. You are allowed to use only AES-128-ECB as a building block.
- Create a Python function that, given the ciphertext, key, IV (as bytes), will decrypt the ciphertext using AES-128-CBC. The function should return the plaintext. Your job is to implement AES-128-CBC encryption using AES-128-ECB as a base. Look for the relevant lecture slides to understand how the algorithm is implemented. You are allowed to use only AES-128-ECB as a building block.
- Create a Python function to verify that both encryption and decryption work, for example by feeding the output of the encryption to the decryption and asserting the output is correct. Also, use the Cryptography library's AES-128-CBC implementation to make sure the encryption is the same. **This function should print any intermediate results**, e.g.: the ciphertext encrypted by either your or Cryptograpy's relevant function, or the result of the decryption.

The script will accept the following command line arguments and pass them to the test function:

- -p: The plaintext, encoded in hexadecimal format
- -k: The key, encoded in hexadecimal format
- -iv: The IV, encoded in hexadecimal format

2. CBC: Random access [10%]

Demonstrate CBC random access property. Specifically, encrypt some blocks (more than 5) of data. Given the generated ciphertext blocks, try decrypting some of the blocks without decrypting the previous ones. With this you should be able to demonstrate that you can access CBC-encrypted data at random locations. **Why is this possible?**

Your function must accept the following parameters:

- The ciphertext, as bytes
- The key, as bytes
- The IV, as bytes
- The index of the block that will be decrypted, as an integer. For example, to decrypt the third block this parameter will be 3.

3. CBC: Man-in-the-Middle (IV = key) [15%]

Assume that in an insecure system, the AES-128-CBC encryption process uses an IV that is equal to the key. The IV and the key are kept secret.

The attacker acts as a man-in-the-middle. We assume they have logged the encrypted message which consists of three blocks (C_1, C_2, C_3). They are able to modify the ciphertext blocks and forward the following sequence to the intended recipient: C_1, 0 (a block consisting of 128 zero bytes), C_1.

The attacker can see the decryption of the three blocks (C_1, 0, C_1) they forwarded earlier. **How is that beneficial to them (what piece of information can they obtain)?**

Implement the attack:

1. Create a random message by using random words from wordlist_1000.txt. Make sure the message is exactly three blocks long by selecting enough words and then truncating the string to fit.
2. Encrypt these three random blocks.
3. Decrypt the modified sequence and then find the required piece of information.
4. Use that to find what the original message was.

4. GCM: Basic functionality [10%]

- Create a Python function that, given the plaintext and key (as bytes), will encrypt them using AES-128-GCM. The function should return the ciphertext, the IV and the tag generated by the algorithm. The IV will be calculated in your function.
- Create a Python function that, given the ciphertext, key, IV and tag (as bytes), will decrypt them using AES-128-GCM. The function should return the plaintext when the decryption is successful or throw an *InvalidTag* exception when the decryption fails.
- Create a main function to verify that both encryption and decryption work.
- This time you CAN use the AES-128-GCM primitives provided by the Cryptography library.

The script will accept the following command line arguments:

- -e/-d: When “-e” is passed, encrypt the plaintext and print the ciphertext, the IV and the tag. When “-d” is passed, decrypt the ciphertext and print the plaintext.
- -t: The text, encoded in hexadecimal format. This can be considered either plaintext or ciphertext, depending on the option above.
- -k: The key, encoded in hexadecimal format
- -iv: If in decryption mode, this will be the IV. If in encryption mode, it can be ignored.
- -g: If in decryption mode, this will be the tag accompanying the data. If in encryption mode, it can be ignored.

5. GCM: Integrity failure in AE mode [10%]

Encrypt some data using the functions from Q4. Try altering any of the encryption elements (i.e., the ciphertext, the IV, the tag or the key). For example, try altering the ciphertext. **What happens to the plaintext? Is any produced? Briefly explain what you observe and how that’s helpful.** Do the same with the rest of the encryption elements listed above, as well as combinations of them (e.g., change both the IV and the tag).

The script will accept the following command line arguments:

- -p: The plaintext
- -k: The key, encoded in hexadecimal format
- -e: The encryption element to damage. It will have to be one of the values “c” (to damage the ciphertext), “iv” (to damage the iv), “key” (to damage the key) and “tag” (to damage the tag). This switch should be able to be given multiple values to damage multiple elements, e.g., “–e key tag” should damage both the key and the tag.

6. GCM: Integrity failure in AEAD mode [15%]

Galois/Counter Mode offers authentication with associated data (AD), meaning that in addition to the plaintext it can verify some extra, unencrypted data. Extend your GCM encryption/decryption functions to handle this scenario.

Try corrupting any part of the encryption output (e.g., the ciphertext or the associated data). **What happens to the plaintext? Is any produced? Briefly explain what you observe and how that’s helpful.** Do the same with the rest of the encryption elements, as well as combinations of them (e.g., change both the AD and the tag).

Hint: Can we use this to transmit something that, while not secret, should arrive at the destination undamaged?

The script will accept the following command line arguments:

- -e/-d/-c: When “-e” is passed, encrypt the plaintext, and print the ciphertext, the IV and the tag. When “-d” is passed, decrypt the ciphertext and print the plaintext. When “-c” is passed, damage the data and the try decrypting
- -t: The text, encoded in hexadecimal format. This can be considered either plaintext or ciphertext, depending on the option above.
- -a: This will be the Associated Data.
- -k: The key, encoded in hexadecimal format
- -iv: If in decryption or corruption mode, this will be the IV. If in encryption mode, it can be ignored.

- -g: If in decryption or corruption mode, this will be the tag accompanying the data. If in encryption mode, it can be ignored.
- -m: The encryption element to damage. It will have to be one of the values “c” (to damage the ciphertext), “iv” (to damage the iv), “key” (to damage the key), “tag” (to damage the tag), or “ad” (to damage the AD). This switch should be able to be given multiple values to damage multiple elements, e.g. “–e key tag” should damage both the key and the tag.

7. GCM: Random access [15%]

Demonstrate GCM random access property. Specifically, encrypt some blocks (more than 5) of data with some IV and key. Given the generated ciphertext blocks, try decrypting some of the blocks without decrypting the previous ones. You can use ECB mode to encrypt the IV with the key and then XOR the result with the ciphertext to get the plaintext. With this you should be able to demonstrate that you can access GCM-encrypted data in a random order. **Is integrity still guaranteed?**

Your code must accept the following parameters:

- The ciphertext, as bytes
- The key, as bytes
- The IV, as bytes
- The index of the block that will be decrypted, an an integer. For example, to decrypt the third block this parameter will be 3.

8. GCM vs ChaCha20-Poly1305: Speed racing [5%]

Using the primitives provided by the cryptography library, speed test GCM and ChaCha20-Poly1305. Generate a large amount of random data (at least 100MB) and feed them to:

- the encryption functions
- the decryption functions

Do this a few times to be sure about any conclusion drawn. Plot the data.

Which algorithm is faster in each case (encryption/decryption)? ChaCha is commonly claimed to be faster. Is this the case? What, in your opinion, affects the results? Consider your CPU. Report the findings.

9. Time-based randomness: Netscape [10%]

In this exercise you will simulate a version of the attack on Netscape's password storage. The encryption key was generated using the code shown below. As the attacker, you can use network tools like Wireshark to estimate the time of the encryption within a minute. That's to say, if the packet's timestamp is 28/10/2023 15:30:23, the encryption happened between 28/10/2023 15:29:23 and 28/10/2023 15:31:23.

```
import random
import time
Import os

seconds = int(time.time())
millisecs = int(time.time()*1000) % 1000
pid = os.getpid() % 10000
seed = seconds*0x68793435382d63727970746f + pid*0x677261706879 + millisecs

r = random.Random(seed)
key = r.randbytes(16)
iv = r.randbytes(12)
```

Create a Python program to decrypt a given ciphertext (encrypted with AES-128-GCM) by trying to guess the key. Your program should take as inputs the estimated Unix time, the ciphertext and the tag. Then, try iterating over the estimated timeframe, the milliseconds and the possible PIDs as seed values for the random function. For each seed value, get a key and an IV, and try decrypting the ciphertext. When no exception is thrown, you have found the key and the IV. Use it to decrypt the ciphertext.

Specifically, the command line parameters will be:

- -d: the estimated encryption time, in Unix format. For the above example, this would be "1698507023"
- -c: the ciphertext to be cracked
- -t: the tag of the ciphertext

10. Bonus: Implement your own GCM [10%]

Using AES-128-ECB as a building block, implement your own GCM encryptor and decryptor. It should be able to handle an arbitrary amount of data, if it is less than 64GiB. Validate your program functions correctly by comparing the outputs of your functions and those provided by Cryptography (or those you made in Q6).

Submission

- Included with the code implementing the questions above, you should provide a README or PDF report file with your answers to the questions in **bold** above and explanations for your implemented or unimplemented parts.

- Submissions will be done through **eLearn**, you will be notified when they open.

- This assignment is an individual creative process and students must submit their own work. You are not allowed, under any circumstances, to copy another person's work. You must also ensure that your work won't be accessible to others.

- You are encouraged to post any questions you may have in the eLearn forum. If, however, you believe that your question contains part of the solution or spoilers for the other students you can communicate directly with the course instructors at *hy458@csd.uoc.gr*.