CSC/ECE 574 - Computer & Network Security

Fall 2023

Mini-Project 1: Crypto

The primary goal of this assignment is to provide an introduction to using cryptographic APIs. Specifically, you will need to specify a secure mode of operation (we are using GCM), correctly generate and use initialization vectors, and ensure both message integrity and confidentiality. You will also be getting first-hand experience in how Diffie-Hellman works, and its susceptibility to on-path attacks.

- Mini-Project 1 is due **Tue, Sep 26** MP1 on or before 11:59:59pm EST.
- The assignment will be submitted via GradeScope. If your GradeScope account was not automatically created and linked, click on the "Mini-Project 1" assignment in Moodle and it should set you up.
- Contact the TA and Instructor if you are having trouble.

Points: Mini-Project 1 has a maximum of 100 points with an additional 10 points for extra credit.

Collaboration:

- You may **not** collaborate on this mini-project. The project should be done individually.
- You **may** search the Internet for help, but you may not copy (either copy-and-paste or manual typing) code from another source.
- You **may** use code from the official Python documentation, PyCryptodome documentation, or from the instructor or TAs.

Posting Solutions: You are explicitly forbidden from posting your solution in a public form (e.g., GitHub). If you need to share your solution as part of a job interview, you should create a private repository and grant that individual access. Please ask the instructor if you have any questions or concerns.

Programming Language: You are expected to use Python 3 for this assignment (not Python 2). Exceptions will be made on a per-student basis in collaboration with the TA (the assignment can not be automatically graded and partial credit may be limited).

- The Python 3 documentation for sockets and PyCryptodome will be very helpful for this assignment.
- Note that while PyCryptodome replaces the no longer maintained (and insecure) PyCrypto module, some source code analysis tools (e.g., bandit) suggest that PyCryptodome should only be used when compatibility with PyCrypto is needed. If you are developing a new project, you are encouraged to use pyca/cryptography which doesn't ask developers to deal with low-level cryptographic primitives. Well … it exposes them through a hasmat API. For the purposes of this assignment, I'd like you to get some experience with the primitives.

Using a Single Host: While we are performing network socket programming, you can test all parts on a single host. Use `localhost` for the destination server and it will work. For Part 4, you will need to specify different ports for the proxy and the server, since two processes cannot listen on the same port on the same host.

Submission

You should submit to GradeScope a `README` text file containing your name and UnityID, as well as the Python 3 source code files for parts 1-5 (5 is optional).

- The **filenames** for the source code files are specified in each part: `uft`, `eft`, `eft-dh`, `dh-proxy`, and `lj-proxy`.
- Note that there is no .py on the ends of these filenames; however, adding .py is okay.

Autograder: This assignment uses an autograder that will automatically grade your work. You will submit your program for autograding to GradeScope.

- Any program that does not have a perfect score will be manually graded after the due date.
- If you find a bug with the autograder, please notify the TA.

Autograder Environment: Your program will be executed on a ubuntu 18.04 docker image with Python version 3.6.

- PyCryptodome is the only additional python package that is installed by default. No additional packages should be required to complete the assignment.
- If your program needs additional packages, you may include a requirements.txt file with your submission and the additional packages will be installed from PyPI using pip before your program is executed.

Part 1 (25 points): Unencrypted File Transfer

- Filename: `utf` or `utf.py`

In Part 1, you will use network sockets to transfer a file between hosts. To simplify operation, the client will read a file from `STDIN` and the server will "save" the file to `STDOUT`. Your code for the client and server **must** reside in the same Python script (`uft`). Your program must differentiate between client and server mode using command line arguments which must conform to the following format:

```
uft [-l PORT] [SERVER_IP_ADDRESS PORT]
```

For example, the following is an example execution.

```
[client]$ ./uft 127.0.0.1 9999 < some-file.txt
[server]$ ./uft -l 9999 > some-file.txt
```

Both programs must terminate after the file is sent. You may assume the server is started before the client.

Important: The program will be executed without an explicit call to the python3 interpreter. To make the program executable, include the following shebang on the first line of your program:

```
#!/usr/bin/env python3
```

Packet Data Unit Structure: The PDUs exchanged between client and server **must** conform to the following specifications to be graded by the autograder:

Data Segment

Element	Size in Bytes	Description	Encoding
Length	2	Number of data bytes following this element	Raw Bytes
Data	Length	File Data	Raw Bytes

Important: All parts of this assignment must work for both small and big files, both text based and binary based. I recommend trying first with a simple text file and then testing with a PDF before submitting.

Tip: I suggest using `sys.stdin.buffer.read()` to read from `STDIN` and `sys.stdout.buffer.write()` to write to `STDOUT`. Both of these functions are available in the `sys` python module.

Tip: The entire file does not need to be sent in a single PDU. Multiple PDUs may be sent, each containing a portion of the file. The autograder will by default send 1024 bytes of data at a time, but it will accept any length up to 65535 bytes.

Tip: Ensure the header bytes are sent in network order (big-endian).

Part 2 (25 points): Encrypted File Transfer

- Filename: `eft` or `eft.py`

In Part 2, you will extend `uft` with symmetric encryption and integrity verification using AES and the Galios Counter Mode (AES-GCM) mode of operation. Recall that GCM avoids the need to incorporate integrity into the cryptographic protocol (e.g., Encrypt-then-MAC).

To perform the encryption, you will use PyCryptodome. Note that PyCryptodome is a drop-in replacement for PyCrypto, which does not support GCM. Unfortunately, most systems provide PyCrypto instead of PyCryptodome, so you may need to read the installation instructions. The documentation for PyCryptodome has several useful examples, but you will likely need to read the API documentation, specifically for using GCM.

You must:

1. Use AES-256 in GCM mode
2. Compute a 32 byte key from the command line argument using PBKDF2 (Password-Based Key Derivation Function), which is available in PyCryptodome. Note that that using PBKDF2 requires a salt, which is a securely generated random value. Both the client and server need to use the same salt; therefore, your connection should start with the client sending the salt to the server. This initial exchange will also get you ready for Part 3.
3. Pad the data into 16 byte (128 bit) AES blocks using the `pkcs7` style. The pad and unpad functions are available as utility functions in PyCryptodome.
4. To successfully decrypt the data, the server must receive the IV ("nonce" in the GCM API) from the client.

Your code for the client and server **must** reside in the same Python script (`eft`), which must conform to the following command line options:

```
eft -k KEY [-l PORT] [SERVER_IP_ADDRESS PORT]
```

The following is an example execution.

```
[client]$ ./eft -k SECURITYISAWESOME 127.0.0.1 9999 < some-file.txt
[server]$ ./eft -k SECURITYISAWESOME -l 9999 > some-file.txt
```

You may assume the server is started before the client.

If an integrity error occurs (e.g., the key is incorrect), the server should write the following error text to `STDERR`. Note that this **exact** error message is required to pass all automated grading checks.

```
Error: integrity check failed.
```

The following is an example execution demonstrating the integrity check output.

```
[client]$ ./eft -k SECURITYISAWESOME 127.0.0.1 9999 < some-file.txt
[server]$ ./eft -k SECURITYISBORING -l 9999 > some-file.txt
Error: integrity check failed.
```

Packet Data Unit Structure: The PDUs exchanged between client and server must conform to the following specifications to be graded by the autograder:

Salt Exchange

Element	Size in Bytes	Description	Encoding
Salt	16	Securely generated random value used in PBKDF2	Raw Bytes

Data Segment

Element	Size in Bytes	Description	Encoding
Length	2	Length of Nonce, Tag, and Data combined	Raw Bytes
Nonce	16	Random Initialization Vector (IV)	Raw Bytes
Tag	16	Integrity Verification Tag	Raw Bytes
Data	Length - 32	Encrypted File Data	Raw Bytes

Tip: PyCryptodome installs into the `Crypto` python module.

Tip: PBKDF2 will by default produce a 16 byte key for AES-128. It accepts an optional length parameter `dkLen` which should be set to 32 to get a 32 byte key.

Tip: Avoid passing other optional arguments to PBKDF2 as this will change the key derived from the shared key which will cause some autograder tests to fail.

Part 3 (25 points): Encrypted File Transfer with Diffie-Hellman Key Exchange

- Filename: `eft-dh` or `eft-dh.py`

In Part 3, you will extend `eft` to calculate a key using the Diffie-Hellman key exchange protocol. Therefore, instead of getting the key from the command line, you will first perform a DH message exchange between the client and the server to establish a symmetric key. The Diffie-Hellman key exchange protocol replaces the PBKDF2 key derivation used in Part 2.

For the key exchange, we will use a fixed `g` and `p` as follows:

```
g=2
p=0x00cc81ea8157352a9e9a318aac4e33ffba80fc8da3373fb44895109e4c3ff6cedcc55c02228fccbd551a504feb4346d2aef47053311ceaba95f6c540b967b9409e9f05
```

You must use a good, cryptographic source of randomness for the DH secrets. Do **not** use Python's `random.random`, PyCryptodome has a secure random number generator. You may also use `os.urandom()` in Python.

Note that Python has native support for handling large numbers (e.g., `pow()` for exponentiation). If you are using C (not supported for the class), you will need `libgmp`.

The output of the diffie-hellman key exchange process should be first encoded to a hex string and then hashed using the SHA256 hashing algorithm. Take the first 32 bytes of output and use it as the session key.

Your code for the client and server **must** reside in the same Python script (`eft-dh`), which must conform to the following command line options:

```
eft-dh [-l PORT] [SERVER_IP_ADDRESS PORT]
```

The following is an example execution.

```
[client]$ ./eft-dh 127.0.0.1 9999 < some-file.txt
[server]$ ./eft-dh -l 9999 > some-file.txt
```

You may assume the server is started before the client.

Protocol Data Unit Structure: The PDUs exchanged between client and server **must** conform to the following specifications to be graded by the autograder:

Diffie Hellman Exchange

Element	Size in Bytes	Description	Encoding
A or B	384	Diffie-Hellman Public value A or B	UTF-8

Important: The Diffie-Hellman Public value A or B is sent as a 384 character UTF-8 encoded string. Zeros must be padded to the left if the output of A or B produces a number that is less than 384 digits in length. The number should not be sent as raw bytes.

Data Segment

Element	Size in Bytes	Description	Encoding
Length	2	Length of Nonce, Tag, and Data combined	Raw Bytes
Nonce	16	Random Initialization Vector (IV)	Raw Bytes
Tag	16	Integrity Verification Tag	Raw Bytes
Data	Length - 32	Encrypted File Data	Raw Bytes

Part 4 (25 points): On-Path Attack on DH Key Exchange

- Filename: `dh-proxy` or `dh-proxy.py`

In Part 4, you will create a proxy called `dh-proxy` that performs an on-path attack on `eft-dh`. To simplify the assignment, we will assume the client connects directly to the proxy and that the proxy connects directly to the target server. Recall from class that an on-path attack is achieved by a) establishing a DH exchange with the client; b) establishing a DH exchange with the server; and c) decrypting data from the client and re-encrypting data to the server. Therefore, you will be able to reuse your DH key exchange code from Part 3.

The tricky part of this part is not the crypto, but rather the network programming. You need to read from the socket with the client and then write to the socket for the server. While you could use threads to handle this, `[select](https://docs.python.org/3/howto/sockets.html#non-blocking-sockets)` is much easier to use.

You **must** conform to the following command line options:

```
dh-proxy -l LISTEN_PORT SERVER_IP_ADDRESS SERVER_PORT
```

The following is an example execution.

```
[client]$ ./eft-dh proxy.ip.address 9999 < some-file.txt
[proxy]$  ./dh-proxy -l 9999 server.ip.address 9998
[server]$ ./eft-dh -l 9998 > some-file.txt
```

You may assume the server is started first, then the proxy, then the client.

Protocol Data Unit Structure: The PDUs exchanged between client, proxy, and server **must** conform to the following specifications to be graded by the autograder:

Diffie Hellman Exchange

Element Size in Bytes Description Encoding

Element	Size in Bytes	Description	Encoding
A or B	384	Diffie-Hellman Public value A or B	UTF-8

Data Segment

Element	Size in Bytes	Description	Encoding
Length	2	Length of Nonce, Tag, and Data combined	Raw Bytes
Nonce	16	Random Initialization Vector (IV)	Raw Bytes
Tag	16	Integrity Verification Tag	Raw Bytes
Data	Length - 32	Encrypted File Data	Raw Bytes

Part 5 (10 Extra Credit Points): Logjam attack on DH Key Exchange

- Filename: `lj-proxy` or `lj-proxy.py`

Part 5 is strictly optional extra credit. I have not completed it, and I don't know how easy or hard it is. However, the idea is to use the logjam attack to eavesdrop on the communication between the client and server **without** performing multiple DH key exchanges. Instead, you should brute force the established key via the logjam attack and write the contents of the transmitted file to `STDOUT`.

Your program must conform to the following command line options:

```
lj-proxy -l LISTEN_PORT SERVER_IP_ADDRESS SERVER_PORT
```

The following is an example execution.

```
[client]$ ./eft-dh proxy.ip.address 9999 < some-file.txt
[proxy]$  ./lj-proxy -l 9999 server.ip.address 9999 > some-file.txt
[server]$ ./eft-dh -l 9999 > some-file.txt
```

You may assume the server is started first, then the proxy, then the client.

Note that since this may be beyond the computational ability of your personal computer, you may modify the `g` and `p` used in `eft-dh`. In this case, provide an alternate `eft-dh-weak` file that has this change. The solution with the largest `p` will receive an additional 5 points.