Assignment 1: Secure Messaging

This project is due on **September 17, 2021** at **5 pm**. You may work with a partner on this assignment and submit one project per team. You may NOT work with the same partner you have worked with on a previous assignment. Submit your solutions electronically.

Background

Secure messaging applications allow sending and receiving messages with guaranteed confidentiality and integrity.

- **Confidentiality:** Messages are not readable by unauthorized individuals
- **Integrity:** Messages cannot be modified in an unauthorized or undetectable manner.

Secure messaging is essential to trusting any Internet communications, but it is not limited to commercial messaging apps or code written by professionals.

For this assignment, you will be creating a secure 2-way messaging application using cryptographic primitives. At the end of the assignment, you will have a Python application that will enable you to exchange secure messages with anyone else in the class. No one else will be able to read these messages (not Colgate ITS, your ISP, other people on the same WiFi network, tech companies, etc.), and you will able to detect any unauthorized message modifications.

It is a bad idea to write your own cryptograhy primitives unless you are an expert ("Don't roll your own crypto!"), so you will be using the publicly available Python libraries PyCryptodome and pyDH.

Objectives

- Gain familiarity with Python socket programming
- Practice reading cryptography library documentation
- Learn to incorporate existing cryptographic primitives into new applications

Ethics Notice

The messaging application you create in this assignment should be secure against unwanted network eavesdropping if implemented correctly. However, cryptography does not protect against end-host compromise, metadata interception, bugs in cryptographic libraries, side-channel inference, court orders, or social engineering. Attempting to use this application for any purpose that is prohibited by law or university policies may result in disciplinary action, expulsion, fines, and/or jail time.

Provided Files

- `Readme.pdf`: This file.
- `SecureMessaging.py`: Starter code with a very simple 2-way INsecure messaging app that uses the Python socket library to create a TCP connection between itself and another instance of the program. One instance of `SecureMessaging.py` must be run as the "client" while the other is run as the "server." Once the connection is made, either side can send and receive string messages up to 2048 characters long.

  Run `SecureMessaging.py` using these commands:

  - Client: `python3 SecureMessaging.py [Server IP] [Server Port]`
  - Server: `python3 SecureMessaging.py [Server Port]`

  A few tips:

  - You may need to allow Python to accept incoming network connections.
  - The server port should be a high unused port (I recommend starting with 10000 and going up from there).
  - If you get an "Address already in use" error, try a different port.
  - If you are running the client and the server on the same computer, use `127.0.0.1` as the server IP.
  - If you are running the client and server on different machines, (e.g. to send messages to your partner or classmates), the person running the server can check their network settings or Google "What's my IP" to find their IP address.

  Once connected, send messages by typing them into the terminal and pressing <enter>.

- `TestMessaging.sh`: Bash testing script to check whether your secure messaging application has the right user interface and can send/receive messages between two instances of itself running on the same machine. Note that the provided `SecureMessaging.py` starter code passes the test, so `TestMessaging.sh` does NOT check whether authentication or encryption has been properly implemented.

  Run TestMessaging.sh using this command: `./TestMessaging.sh [server-port]`

  A few tips:

  - The first time you run TestMessaging.py you may need to make it executable using this command: `chmod 764 TestMessaging.sh`
  - The server port should be a high unused port (go up from 10000).
  - If you get an "Address already in use" error, try a different port.
  - `TestMessaging.sh` is somewhat brittle, and doesn't always work depending on your operating system. Ultimately, being able to send messages back and forth using the normal python interface is more important than the `TestMessaging.sh` results.

- `Questions.txt`: Written questions. Fill in your answers in this file and submit it.

Instructions

1. Install python3 from `python.org` if you do not already have it on your computer.

2. Install the PyCryptodome and pyDH libraries by running the command `pip3 install pycryptodome pyDH` from your terminal.

3. Review and try out `SecureMessaging.py` starter code, making sure you understand the difference between the "client" and "server" initialization and how messages are sent and received. If this is your first experience with socket programming, you may want to review the documentation at `https://docs.python.org/3/library/socket.html`. If you have additional questions about how the provided `SecureMessaging.py` works, post a question on Slack, email Prof. Apthorpe, or talk to a TA.

4. Negotiate protocol details with your classmates on Slack. In order to communicate with everyone in the class, your secure messaging applications will need to use consistent message formats and a communication protocol that answers the following questions, as well as others that may arise during implementation:

   (a) What will be the format of Diffie-Hellman key exchange messages?

   (b) How will message authentication codes be included (and differentiated from) the content of messages?

   (c) In what order will MACs (for integrity) and encryption (for confidentiality) be applied to messages?

   (d) What settings options will you use for D-H key exchange and the AES cipher?

   While you must propose and discuss potential protocols and message formats with your classmates, you MUST NOT share code specifics with anyone other than your partner and Prof. Apthorpe. Imagine that you are startups creating messaging apps: you need a standard protocol so your users can communicate across platforms, but you don't want to share source code with the competition.

5. Modify `SecureMessaging.py` to include the following required features while adhering to the protocol agreed upon with your classmates:

   (a) Use the PyCryptodome library to generate all needed random numbers. Documentation here: `https://pycryptodome.readthedocs.io/en/latest/src/introduction.html`

   (b) Use the pyDH library to implement Diffie-Hellman key exchange upon client/server connection. Before any user messages are sent, the client and server should have **at least two shared keys**, for sending messages in client $\rightarrow$ server and server $\rightarrow$ client directions. Documentation here: `https://github.com/amiralis/pyDH`

   (c) Use the PyCryptodome library to add authentication to messages with HMAC-SHA256 and confidentiality to messages by encrypting with an AES block cipher.

   (d) If you detect that a message has been modified in transit (via the HMAC), you should raise a `ValueError("MessageModificationDetected")`

(e) Your modifications must **not** change the user-facing behavior of `SecureMessaging.py` You should still be able to run `SecureMessaging.py` using the following commands and send messages by typing them in to the terminal and pressing <enter>:

- Client: `python3 SecureMessaging.py [Server IP] [Server Port]`
- Server: `python3 SecureMessaging.py [Server Port]`

`SecureMessaging.py` must not require any additonal input from the user to provide authentication and encryption.

6. Test your modified `SecureMessaging.py`

(a) By sending messages to your partner and to your classmates. Grading will be based on whether your `SecureMessaging.py` can communicate with another instance of itself and that you correctly implemented the protocol agreed upon with your classmates on Slack. You do **not** need to be able to communicate with every one of your classmates' implementations for full credit (as their implementations might be incorrect).

(b) With the provided `TestMessaging.sh`

(c) *[Optional]* For a small amount of extra credit, try writing your own testing code to verify that your `SecureMessaging.py` works as expected.

7. Answer the questions in `Questions.txt`

Deliverables

One partner should submit the following files to Gradescope:

- `SecureMessaging.py` with all modifications from Instruction #5
- `Questions.txt` completed with your **concise** answers
- *[Optional]* `NewMessagingTests.py` with your custom extra credit testing code

Extra Credit "Bug Bounty"

If you find a bug anywhere in this assignment, please inform Prof. Apthorpe. The first student (or partners) to find any particular bug will be given a small amount of extra credit. This is an incentive to start the assignment early and will help make the course better for students in future years.