CMPSC 443: Introduction to Computer Security
Fall 2025 Project 1: Cryptography
Due: 11:59 pm (Eastern Time), September 26, 2025

General Instructions

There are **three tasks** in this project, each with many sub-tasks. There is one optional **bonus** question (task-1b), which will be awarded with bonus points that add up to your score.
The maximum possible score in this project will be **270 points**. Out of which 250 points constitute 100% and there is a scope for 20 bonus points.

To complete all the tasks in this lab, we recommend using a Linux-based operating system. While we have tested the exercises in macOS and Windows, they might need additional setup for some tasks.
If you want to use Windows, set up "minGW" and run all commands in it. This is required for the "OpenSSL" application to be installed.
Your submission should include the following: a PDF for the report (preferably written using LaTeX), the folder for task 3 containing the three Python scripts. You should make a zip of all the items.

Write how you did your project in your report. Be ready for the Demo! Keep your project environment ready for executing one or more tasks for the course staff. Do NOT use generative AI to generate the code! You will be asked to explain your code or design choices during the Demo!

Start your project as soon as you can; it can be time-consuming, especially for task 3.

Task 1 – *(50 points + 20 Bonus points)*

a. Frequency Analysis - 1 *(50 points)*

In this task, you are given a ciphertext encrypted with a mono-alphabetic substitution cipher. It means each English letter will be replaced by another letter. In the class, you have learned that this kind of encryption is weak and can easily be cracked by frequency analysis. Your task is to perform frequency analysis on the given ciphertext (ciphertext.txt) and try to decipher the original article.

You can use some online tools or the provided Python program (freq.py) to perform the frequency analysis.

freq.py usage: `python3 ./freq.py <filename>`

After determining the frequency of letters in the given file, you will need to check the common frequency in the Online Resources.

Now you may want to change some letters in the ciphertext back to plaintext. For example, if you want to replace the letters a, c, and f in in.txt with letters Z, K, and J and output the result to out.txt, you can use the "tr" command as follows in Linux:

```
$ tr 'acf' 'ZKJ' < in.txt > out.txt
```

You may be able to guess some words when most letters in that word are decoded. Doing so, you will find more clues in the decoding process.

Submission:
You need to write all the steps you did, the substitution key, and the decoded text in the report. You may need to attach screenshots to help explain the steps.
Online resources:
https://en.wikipedia.org/wiki/Frequency_analysis
https://en.wikipedia.org/wiki/Bigram
https://en.wikipedia.org/wiki/Trigram

Notes:

- Some letters may not appear in the ciphertext; you do not need to worry about them.
- We recommend that you use uppercase letters to substitute ciphertext. Since the ciphertext will only contain lowercase letters, it is easier to distinguish between original letters and substituted letters.
- The frequencies in our ciphertext may be different from the common frequencies.

[BONUS] b. Frequency Analysis – 2 *(20 points)*

In this part, you are given a ciphertext encrypted with Vigenère cipher. The ciphertext provided is encrypted with a 3-letter English word. You will need to perform frequency analysis to the ciphertext and find out the key.
Feel free to check the Wikipedia page below to learn more about Vigenère cipher.
https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher
Cipher Text:

```
kb lhw kjm qlwk qn ipqvu, qi dej vpt dsiub dm xzomh, px nca ioi rim dm azuldt, mk yih alv cot vj wqwapwypmhz.
```

Notes:

- The most common letters in English are E, T, A, O, and I.
- Group letters encrypted by the same letter in the key so that you can do the correct frequency analysis.
- The key is a legal 3-letter English word, and the decrypted sentence is also a legal English sentence.

Submission:

- You need to write all the steps you did, the substitution key, and the decoded text in the report. You may need to attach screenshots to help explain the steps.

Task 2 – *(30 points)*

Hashing is the process of converting plain text secret data into another value that can be sent to a verifier to verify the authenticity of the sender. A good hash function uses a one-way hashing algorithm, or in other words, the hash cannot be converted back into the original key.
MD5 and SHA-1 are examples of hashing algorithms that are now considered too weak for secure hashing purposes. Hence, many organizations have deprecated these algorithms for hashing.

a. Plain Hashing – 1 ***(10 points)***

Following is the hash generated for some secret phone number –
`0e440f7fb09e64be1c535bb3e22c978e`
Find the original text that was used to generate this hash. Report the original text, the hashing algorithm used, and how you found the same!
Hint: it should be a 9-digit number.

b. Plain Hashing – 2 ***(10 points)***

Following is the hash generated for some secret data –
`867230eae0f2ad45b63b9bba390b3e231ce0202f`
Find the original text that was used to generate this hash. Report the original text, the hashing algorithm used, and how you found the same!

c. Hashing – 3 ***(10 points)***

From the previous steps, it's evident that simple character strings and easy passwords can easily be "un-hashed". Without using stronger passwords or stronger hashing algorithms, can you suggest one technique to build stronger Hashes that cannot be easily un-hashed? Explain.

Task 3 – *(170 points)*

Imagine you're building a secure cloud storage service called 'Crypto443 company'. This means even you, the company, can't access the user's files. The only person who can is the user themselves.

The Problem: Users need to upload and download highly sensitive documents like financial records or medical files. They don't trust you, the company, to keep their data private, especially from potential government subpoenas or rogue employees.

So you propose several approaches to make this secure

- The communication between server and client is encrypted and authenticated.
- The password-based authentication is secured using Secure Remote Password. This ensures that the user's password is never leaked.
- The client-side encryption of files ensures that the data is encrypted before it ever leaves the user's computer. The server only ever receives and stores the unreadable ciphertext.
  Meaning that even if the server is breached, the attacker will only find encrypted files, rendering the data useless without the user's password.

We use the cryptography library for Python (which uses OpenSSL under its hood):
https://cryptography.io/en/latest/

Overall structure of the project
The file you are given looks like this

```
├── client_folder
│   ├── downloaded
│   │   └── test.txt
│   └── filestosend
│       └── test.txt
├── client.py
├── server_folder
│   ├── server_files
│   │   └── user_id_1
│   └── valid_ids.txt
├── server.py
└── util.py
```

test.txt here is simply a placeholder; it could be replaced with anything.

The current communication looks like this: (at the directory)
First, run `python server.py`, then run `python client.py` in another terminal window.
The client should be interactive. You can sign in/up by providing the ID and password.

The server would sign up if the ID is new; if the ID is registered, then the server should match the password and reject any incorrect password. The list of users should be stored in server_folder/valid_ids.txt.

Now, once the client is signed in, it has 3 choices: `send/get/exit`.

For `send`, it will send a file in the folder client_folder/filestosend, the server should receive the file, and store it in the directory server_folder/server_files.

For `get`, it will retrieve the file from the server. If the file is not present, the server will return an ERROR. If it is present, the server should send the file to the client, and the client should store the file in the client_folder/downloaded.

For `exit`, it will simply exit.

A possible scenario is as follows:

1. Start server and client programs
2. Register new user ‘alice’ with password ‘randomword’
3. Now you type ‘send test.txt’, the server should securely store the test.txt
4. Restart the server and client
5. You should be able to log in as the existing user ‘alice’ with the same password
6. You should be able to retrieve the file by ‘get test.txt’, it should be stored at ‘client_folder/downloaded’
7. Restart the server and client
8. You should not be able to log in if you use ‘alice’ with a different password
9. Restart the server and client
10. You should be able to register new users/login, and send/get files.

Some caveats: the server can only authenticate one client at a time. If you want to start a new session, exit both programs first.

Now, your task is to make it more secure. You will need to modify the client.py server.py and util.py.

Step 1. Establish Secure Channel (Key Exchange) (70 pts)
In this step, the server and client will establish a secure connection. This is done by using a combination of asymmetric and symmetric encryptions. This step is important because then an attacker could not learn or alter the information exchanged by the client and the server.

Current: Placeholder for key exchange; keys are not actually exchanged securely.
Requirement:

- Implement public-key cryptography for key exchange:
  The server should create a pair of asymmetric keys. And send the public key to the client.
  (in this case, we suppose this step is secure, that the client can obtain the public key without anyone altering it.)

More specifically, you should use RSA as follows

`cryptography.hazmat.primitives.asymmetric.rsa.generate_private_key` to generate the keys.

- Now, the client should generate a symmetric key, named session_key, for secure communication. For this communication, you should use Authenticated Encryption, ChaCha20Poly1305 in this case.
  Use
  `cryptography.hazmat.primitives.ciphers.aead.ChaCha20Poly1305.`

- Next, the client should send the session_key to the server using the public key. (You can use any available padding as you like)
- The server decrypts the session_key with its private key.
  Now, both sides use the session_key for all further communication.
- Report how you accomplished these steps.

Step 2. Secure Password Management. (40 pts)
In this step, you will implement password-based authentication using the Secure Remote Password (SRP) protocol.
Reference: https://en.wikipedia.org/wiki/Secure_Remote_Password_protocol

SRP allows a client (the user) to prove knowledge of a password to a server without ever sending the password itself, nor any value from which the password can be directly derived. The password remains secret on the client side and is never revealed to the server.

You will use the srp Python package for this task. https://pythonhosted.org/srp/srp.html
It should already expose a simple interface that you can use to authenticate the password.
You should read how to use the srp package, and come up with your design of the protocol.

Requirements:
- The client must not send the password during authentication.
- Instead, the client computes and sends only the salt and verifier to the server during registration. You can use the default algorithms when using create_salted_verification_key()
- The server should store a list of valid clients (with their salts and verifiers) in the file: server_folder/valid_ids.txt
- The server should authenticate existing clients.
- For this step, you do not need to derive or use a new symmetric session key. Authentication alone is sufficient.
- Report your design choices.

Step 3. Secure File Encryption. (60 pts)
Current: Client sends the file in plain text, and the server stores it in plain text.

Requirement:

- You should encrypt and authenticate your file on the client side, since the server is not trusted with the content of your files.
- You should use `cryptography.hazmat.primitives.ciphers.aead.AESCCM` for authenticated encryption.

- Your file encryption and authentication keys should be derived from the user's password. This is the reason why you should never share the password with the server!

  Moreover, these keys should be salted, and you should produce a new salt for each file. You should not reuse the salt, but remember you can store the salt on the server.

  You should use `cryptography.hazmat.primitives.kdf.pbkdf2.PBKDF2HMAC` to derive the file encryption keys.
- After you have encrypted the file, you should still communicate this encrypted file through the secure channel we established earlier. (send_data/receive_data function in util.py)

- Another problem you should note: different clients might send files with the same name. You should distinguish between them by creating a separate directory for each client. Moreover, the same client should overwrite the old file if the new file has the same name. However, in our system, we do not care about the secrecy of the file names nor the file length.

- Report how you manage and derive the keys, how you encrypt and decrypt the files.

Note:

- You can earn partial credit for successfully completing individual parts or steps of the project.
- However, to receive any of that credit, the overall program (even if some parts are still the basic, unmodified version without the advanced security features) must be functional and executable. If the program doesn't run at all, you won't get credit for the parts you might have completed.
- You should not attempt to trick the autograder.