CS 352 Assignment 3
HTTP Server with User Authentication and File Serving

Disclaimer: This project is meant for educational purposes only. A real HTTP server (such as Apache) has been developed over decades by hundreds of experienced developers.

A note: Blindly using GPT and friends to produce code *will probably* get you flagged for plagiarism. The plagiarism detector will check against code structures generated by GPT.

Project description:

You will create a small HTTP server that provides basic user authentication and supports returning contents of files. Users can log in with a username and password, and upon successful authentication, they can query the contents of files from the users directory. The server is designed to use cookies to manage user sessions and to ensure the security of file retrieval.

Server functionality:

- User Authentication:
  - Users can log in with a username and password.
  - Passwords are stored as SHA-256 salted-hashed values in an “accounts.json” file.
  - Successful logins are authenticated by hashing the plaintext password with the salt stored in “accounts.json”.
  - User sessions are tracked using a cookie with a timeout.
  - Multiple users can be logged in at the same time.

- File Download:
  - Authenticated users can view the contents of files from a specified directory.
  - File access is restricted to the authenticated users directory only.
  - Unauthorized access to files is denied.

- Session Management:
  - User sessions are managed using randomly generated session IDs stored as a cookie.
    - Sessions can be tracked using a Python dictionary, which maps session IDs to usernames and login times.
  - Sessions expire after a configurable timeout period.

Server implementation:

The server should be implemented in Python3, using TCP sockets to handle incoming HTTP requests. HTTP requests should be parsed manually, without the assistance of any external modules (see recitation 9 slides). Use of external modules may cause the autograder to give 0 points.

Allowed Python modules:

- `socket, json, random, datetime, hashlib, sys`

The server is executed with command-line arguments, specifying the IP address, port, accounts file, session timeout, and the root directory for file downloads.

Usage:

Your server should process only the following command-line arguments:

```
python3 server.py [IP] [PORT] [ACCOUNTS_FILE] [SESSION_TIMEOUT] [ROOT_DIRECTORY]
```

Example: `python3 server.py 127.0.0.1 8080 accounts.json 5 accounts/`

- `IP`: The IP address on which the server will bind to.
- `PORT`: The port on which the server will listen for incoming connections.
- `ACCOUNTS_FILE`: A JSON file containing user accounts and their hashed passwords along with the corresponding salt.
- `SESSION_TIMEOUT`: The session timeout duration (in seconds).
- `ROOT_DIRECTORY`: The root directory containing user directories.

Included project files and scripts:

You are provided with the following goodies:

- `server.py`
  - This is where you will implement all of your code for the HTTP server.
- `passwords.json`
  - This json file contains key-value pairs of username-plaintext_password combos for all existing user accounts. This file will be used by the client to login to an account. The username and plaintext passwords should be interpreted as strings.
- `accounts.json`
  - This json file contains key-value pairs of username-[hashed_password,salt] combos for all existing user accounts. Note that the value for a username key is a json array where the first element is the hashed salted password and the second

element is the salt itself. This file should be used by the server to validate login credentials. The username and password with the salt are stored as strings.

- `sample.sh`
    - This script contains some CURL commands for local testing of your HTTP server.
- `accounts/`
    - This directory contains user accounts and files accessible by the server.

Server logs:

Your server should log (print) login attempts, file downloads, and session expirations, including timestamps. The format is as follows:

```
SERVER LOG: [current time as Year-month-day-hour-minute-second] [MESSAGE]
```

The MESSAGE field of the logged output is specified in the pseudocode for the server. Here is an example of what is printed on a successful login for the user “Jerry”, in pseudocode we refer to this as: **`log with MESSAGE "LOGIN SUCCESSFUL: {username} : {password}”`**

```
SERVER LOG: 2023-10-31-10-13-45 LOGIN SUCCESSFUL: Jerry : 4W61E0D8P37GLLX
```

Server Specifications:

HTTP and Requests:

- The server can use HTTP version 1.0, but the server should not care what HTTP version the client uses.
- A POST request is used for logging in. The request must have a request target of “/”
- A GET request is used for retrieving files after logging in. The request must have a request target of the filename from the root “/”, such as “/file.txt”.
    - The server then finds the “file.txt” file in the directory for that user only (via the username)
    - The server must read the contents of the file as text and insert it into the HTTP body of the response
- A sample POST request with response
    - Client sends:

```
POST / HTTP/1.0
Host: 127.0.0.1:8080
User-Agent: curl/7.68.0
Accept: */*
username: Jerry
password: 4W61E0D8P37GLLX
```

- Server replies with:

```
HTTP/1.0 200 OK
Set-Cookie: sessionID=0x68938897ef8fdfc8

Logged in!
```

- Server logs:

```
SERVER LOG: 2023-11-02-15-16-46 LOGIN SUCCESSFUL: Jerry : 4W61E0D8P37GLLX
```

- A corresponding sample GET request for the file “file.txt” with response. Note: This is after logging in with the previous POST request and still within the session timeout limit.
  - Client sends:

```
GET /file.txt HTTP/1.0
Host: 127.0.0.1:8080
User-Agent: curl/7.68.0
Accept: */*
Cookie: sessionID=0x68938897ef8fdfc8
```

  - Server replies with:

```
HTTP/1.0 200 OK

The different snowstorm exhibits fee.
```

  - The server also logs:

```
SERVER LOG: 2023-11-02-15-23-21 GET SUCCEEDED: Jerry : /file.txt
```

Cookies:

HTTP is a stateless protocol, and thus some data needs to be sent on each request to maintain a state (i.e. user session). For this, HTTP has a header field called “Cookie” which contains a list of key-value pairs for each cookie. When using a browser, cookies are automatically sent and maintained per website domain. Since we are not using a browser to access the HTTP server in this project, we will manually send cookies on each request. To first obtain a cookie, the server must send back the header “Set-cookie” which tells the client (i.e. browser) to set a cookie in the “Cookie” header.

Cookie Format:

The cookie should be called ”sessionID” and should consist of a random 64 bit integer in string hexadecimal format. Here is an example of the “Set-cookie” header the server may return:

```
Set-Cookie: sessionID=0xfff3c577d6381f1d
```

How to handle passwords and salts in the password file:

Storing a plaintext password leads to security flaws in systems. What happens if the password store is stolen, such as the case of a database breach? In order to authenticate passwords without actually storing them, servers can ***hash*** the password and store the hash. Authentication then works as follows:

1. The client sends the username and plaintext password to the server using a POST request.
2. The server hashes the plaintext password.
3. The server performs a lookup into its database to check if the hashed password corresponds to the username supplied.
4. The server either logs in or fails respectively.

But this method of storing passwords is still insecure. *What happens if an attacker steals the database of hashed passwords?* It may appear our plaintext passwords are safe, as the attacker cannot reverse the hashes. Although this is true, the attacker can perform a *dictionary attack*; this is when the attacker hashes a list of commonly used words and checks if these hash to any of the hashes in the stolen database. For example, many people may have the “apple” as a password. As such, many of the hashes in the database would be hash(“apple”), and the attacker could find these. On the other hand, if a user chooses a random string, such as “EDVcS” as a password, then it is unlikely the attacker would choose such a plaintext password to hash for a dictionary attack. Of course we would like this property to hold for regular passwords such as “apple”, and for this we will use a ***cryptographic salt***

A salt is a random piece of data appended to the password string by the server before it is hashed. Since a small change in input leads to vast changes in output for cryptographic hash functions, a salt that is appended to a plaintext password will hash as if the password itself was random. This is what we want; if many people pick the password “apple” we can generate a random salt for each user and append it to the end of “apple” and hash this instead. The plaintext salt is then stored alongside the hashed password + salt combo. A dictionary attack now requires adding each stored salt to the end of phrases to correspond the hashes, preventing hash(“apple”) from being a common hash among the leaked data.

HTTP Format

HTTP is an application layer protocol that usually consists of UTF-8 characters. HTTP messages consist of a series of lines ending in a CRLF (“\r\n”). HTTP message lines consist of a start-line, zero or more header fields (also known as "headers"), an empty line (i.e., a line with nothing preceding the CRLF) indicating the end of the header fields, and possibly a message-body. This means you can parse an HTTP message in Python by just reading the lines from the TCP socket. Detailed information about the formatting of an HTTP message can be found here: https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages

Parsing Json

A JSON file can be turned into a Python dictionary with a couple lines of code if you use the `json` module. A simple example can be found here:
https://www.geeksforgeeks.org/convert-json-to-dictionary-in-python/

Pseudocode for Server:

```
Import necessary libraries

Function to handle a POST request for user login:
    Obtain "username" and "password" from request headers
        If 1 or both fields missing:
            Return HTTP status code "501 Not Implemented", log with
            MESSAGE "LOGIN FAILED"
    If "username" and "password" are valid:
        Set a cookie called "sessionID" to a random 64-bit
        hexadecimal value
        Create a session with required info for validation using the
        cookie
        Log with MESSAGE "LOGIN SUCCESSFUL: {username} : {password}"
        Return HTTP 200 OK response with body "Logged in!"
    else:
        Log with MESSAGE "LOGIN FAILED: {username} : {password}"
        Return HTTP 200 OK response with body "Login failed!"
Function to handle a GET requests for file downloads:
    Obtain cookies from HTTP request
    If cookies are missing return HTTP status code "401 Unauthorized"
    If the "sessionID" cookie exists:
        Get username and timestamp information for that sessionID
        If timestamp within timeout period:
            Update sessionID timestamp for the user to current time
            If file "{root}{username}{target}" exists:
                Log with MESSAGE "GET SUCCEEDED: {username} : {target}"
                Return HTTP status "200 OK" with body containing
                the contents of the file
            Else:
                Log with MESSAGE "GET FAILED: {username} : {target}"
                Return HTTP status "404 NOT FOUND"
        Else:
            Log with MESSAGE "SESSION EXPIRED: {username} : {target}"
            Return HTTP status "401 Unauthorized"
    Else:
        Log with MESSAGE "COOKIE INVALID: {target}"
        Return HTTP status "401 Unauthorized"

Function to start the server:
    Create and bind a TCP socket
```

```
    Start listening for incoming connections
    While True:
        Accept an incoming connection
        Receive an HTTP request from the client
        Extract the HTTP method, request target, and HTTP version
        If HTTP method is "POST" and request target is "/":
            Handle POST request and send response
        Elif HTTP method is "GET":
            Handle GET request and send response
        Else:
            Send HTTP status "501 Not Implemented"
        Close the connection

Function Main:
    Call the start server function and pass command-line arguments
```

Using Curl as the HTTP client

Test cases used by autograder

1. No Username (POST at the root)
2. No Password (POST at the root)
3. Username incorrect (POST at the root)
4. Password incorrect (POST at the root)
5. Username (1st username) correct/password correct (POST at the root)
6. Username (1st username) correct/password correct (POST at the root) -> Generate a new cookie
7. Invalid cookie (GET)
8. Username (1st username) (GET filename for user 1) correct
9. Username (2nd username) correct/password correct (POST)
10. GET file successful (GET filename for user 2)
11. GET file not found (GET FAIL)

Sleep for 6 seconds

12. Expired cookie with username 2 (GET filename for user 2)

Every test is worth 10 points for 120 points total.

Sample Curl commands:

Provided in `sample.sh`

Handing in Assignment:

You need to upload your code to Gradescope. Log into canvas and use the Gradescope tool on the left. The assignment is called “HTTP Server”. Upload a Python file called `server.py`.

How to add group members in gradescope:
https://help.gradescope.com/article/m5qz2xsnjy-student-add-group-members

References:

- About HTTP: https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages
- HTTP Format: https://docs.netscaler.com/en-us/citrix-adc/current-release/appexpert/http-callout/http-request-response-notes-format.html#format-of-an-http-request
- Python Socket Programming: https://docs.python.org/3/library/socket.html
- Python hashlib Module: https://docs.python.org/3/library/hashlib.html
- Cookies: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies
- Curl tutorial: https://curl.se/docs/tutorial.html