

Project #3: HTTP Server with Authentication

CS 352 Internet Technology

Released: February 19, 2023; Due: March 09, 2023

Instructions

Please read these instructions carefully before you begin.

1. You must work on this project in teams of 2.
2. You are free to discuss the project on Piazza or through other means with your peers and the instructors. You may refer to the course materials, textbook, and resources on the Internet for a deeper understanding of the topics. However, you cannot lift solutions from other students or from the web including GitHub, Stack Overflow, or other resources. Do not post this project to question-answering services like Chegg. All written and programmed solutions must be your team's original work. We run sophisticated software to detect plagiarism and carefully monitor student answers. If you are in doubt, please ask us.
3. You cannot post your solutions to this project on your personal GitHub page or on other web services hosting class materials.
4. For each question in the project report, please be clear and concise. Vague and rambling answers will receive zero credit.
5. For the report question on collaboration, please include anyone you discussed the project with, and also any resources you consulted to learn how to solve this project, including URLs of pages visited on the Internet. Please be specific about the aspect of the project that you got help with. You must be thorough and as complete as possible here. It is mandatory to answer this question.
6. We encourage you to *start early* and get the bulk of the work for this project done the week(s) before it is due, rather than keeping it until the submission date.
7. If you have any questions or clarifications on the project, please post them on Piazza or contact the course staff. We are here to help.

Overview

In project 3, you will implement an HTTP server that serves secret user data. Access to this secret data will be authenticated through one of two mechanisms: (i) user name and password, and (ii) cookies presented from successful prior authentication. We will use the HTTP protocol and build simple versions of these authentication mechanisms to render browser-readable data. You will only work with one program in this project, `server.py`.

Step 1: Let's take it for a spin!

The project archive comes with starter code for `server.py`. This file already implements an HTTP server which serves a login page. Start the server program by either typing

```
python server.py
```

which spins up the server on port 8080 (default), or you can type a port number of your choice:

```
python server.py 45006
```

The best part about the HTTP protocol is that you can interact with the other endpoint using your browser. Spin up the local browser and type `http://localhost:45006` (substitute 45006 by the port number you used when you started the server, which may be the default value 8080.)

You may see something that looks like this:

Please login



Name:

Password:

You can also interact with the server using a command line client like `curl`. You could type the command `curl http://localhost:45006/` to perform a transaction through the command line. You will see the HTML content of the page printed on the terminal. You can get `curl` to print more information using the `-v` flag. Here is an example of what you may see when you invoke `curl -v http://localhost:45006/`

```
* Trying ::1...
* TCP_NODELAY set
* Connection failed
* connect to ::1 port 45006 failed: Connection refused
* Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 45006 (#0)
> GET / HTTP/1.1
> Host: localhost:45006
> User-Agent: curl/7.54.0
> Accept: */*
>
```

```

< HTTP/1.1 200 OK
< Content-Type: text/html
* no chunk, no close, no size. Assume close to signal end
<
<h1>Please login</h1>
  <form action = "http://localhost:45006" method = "post">
    Name: <input type = "text" name = "username"> <br/>
    Password: <input type = "text" name = "password" /> <br/>
    <input type = "submit" value = "Submit" />
  </form>
* Closing connection 0

```

You can ignore the error message at the beginning from the IPv6 connection attempt. If you go back to the terminal running the server program, it should have printed some helpful messages regarding the request it received and the responses it sent out. Example:

```

Here is the headers
"""
GET / HTTP/1.1
Host: localhost:45006
User-Agent: curl/7.54.0
Accept: */*
"""

```

```

Here is the entity body
"""

"""

```

```

Here is the response
"""
HTTP/1.1 200 OK
Content-Type: text/html

```

```

<h1>Please login</h1>
  <form action = "http://localhost:45006" method = "post">
    Name: <input type = "text" name = "username"> <br/>
    Password: <input type = "text" name = "password" /> <br/>
    <input type = "submit" value = "Submit" />
  </form>

"""

```

```

Served one request/connection!

```

Step 2: Study `server.py` carefully

In many ways, `server.py` is not that different from any of the TCP servers you implemented in project 1 or project 2. The key difference is that it receives HTTP protocol request messages and responds with HTTP protocol response messages.

It will be worthwhile understanding how `server.py` unpacks the data in the HTTP request to extract the headers and the entity body of the request. You will build on this code to extract specific headers and information from the entity body.

The `server.py` code also contains various strings that are helpful to return as entity bodies of the HTTP responses, such as `login_page`, `bad_creds_page`, and so on.

Pay specific attention to the places marked `TODO;`; these are the locations you will insert the main application logic for this project.

Step 3: Building the databases

The server should read all of the username and passwords stored in the file `passwords.txt` (in plain text) provided along with this project archive. This file will contain multiple lines, one per user of the system, with their username and password (always one word) in that order, separated by a space. For example, in the sample passwords provided, one of the users of the system is named `bezos` with the password `amazon`. Your program should read in all this user data and maintain a data structure for lookup when the user attempts to authenticate.

The server should also read in the corresponding user secrets stored in plain text in the file `secrets.txt`, provided along with the project archive. This file will contain multiple lines, one per user of the system, with their username and secret data (always one word) in that order, separated by a space. For example, in the sample secrets provided, one of the users of the system is named `bezos` with the secret word `kaching`. Your program should read in all this user data and maintain a data structure to lookup for displaying after successful authentication.

Step 4: Implementing username-password authentication

The browser login form on `http://localhost:45006/` is set up to send POST requests when these details are typed into the respective fields. (Beware, the file is stored in plaintext, and any information you type in the form is sent in plain text, so please do not put any sensitive information in these!)

You can also send POST data from the command line using the `curl` tool. If you type the command

```
curl -d "username=bezos&password=amazon" http://127.0.0.1:45006/
```

it has the same effect as form data posted from a browser window. The `server.py` terminal output should show

```
Here is the entity body
"""
```

```
username=bezos&password=amazon
"""
```

in place of the (empty) entity body from before. Specifically, the command flag `-d` makes `curl` send a POST request with the data inserted into the entity body of the HTTP request.

Parse the entity body obtained by `server.py` to recover the username and password fields of the HTTP POST request. Compare these details with the information you read from `passwords.txt` in the previous step. Not all valid HTTP requests may contain these two fields in their entity body, so you must pay careful attention to handling different kinds of requests in your code.

Case A: Username-password auth success. If the username and password fields do exist in the entity body, and they match with the username and password of a user in the passwords file, you can return the `success_page` to this user with the corresponding secret information that you read in the previous step. An example of this output on the browser looks like this, after `bezos` successfully authenticates:

Welcome!

[Click here to Change Password](#)

[Click here to logout](#)

Your secret data is here:

kaching

To accomplish this in the code, you merely need to set the variable `html_content_to_send` to `success_page + secret`, where `secret` is the secret word of the corresponding user who just logged in.

Case B: Username-password auth failure. If exactly one among the username or password fields is absent in the entity body (i.e., exactly one field is present), or if both fields are present but the username is not in the passwords file, or the password did not match the corresponding username in the passwords file, then we ask the user to log in again. This is accomplished by setting `html_content_to_send` to `bad_creds_page`. You may see output like this:

Bad user/pass! Try again

Name:

Password:

You should be able to see HTML source code corresponding to these pages using appropriate `curl` commands as well, in case you want to test your program quickly on the terminal.

Step 5: Generate, send, and store a cookie

Now we get to the part where the server “remembers” prior successful authentications using cookies. Recall that cookies are a collaborative mechanism between the client and the server. The server assigns an opaque identifier to a successfully authenticated user, and sends it back in a `Set-Cookie` header in the HTTP response.

Step 5.1 In case A in the previous step (successful username-password authentication), generate a cookie value which is a random 64-bit value. You can accomplish this by invoking `random.getrandbits(64)` in your code. Capture this header-value pair in a string that contains the entire HTTP header line. If you assign `headers_to_send` to this string, it will be sent to the client along with the rest of the message. An example of doing this is:

```
rand_val = random.getrandbits(64)
headers_to_send = 'Set-Cookie: token=' + str(rand_val) + '\r\n'
```

(The `token=` is helpful to have `curl` record and reuse (present) cookies, as we will see later.) An example output of the full HTTP response in this case from the terminal output of `server.py` looks like this:

Here is the response

```
"""
```

```
HTTP/1.1 200 OK
```

```
Set-Cookie: token=24014456
```

```
Content-Type: text/html
```

```
<h1>Welcome!</h1>
```

```
<form action="http://localhost:45006" method = "post">
```

```
<input type = "hidden" name = "action" value = "logout" />
```

```
<input type = "submit" value = "Click here to logout" />
```

```
</form>
```

```
<br/><br/>
```

```
<h1>Your secret data is here:</h1>
```

```
kaching
```

```
"""
```

Step 5.2 Store the cookie `rand_val` that you sent in step 5.2 into another data structure that can look up the cookie to obtain the corresponding pre-authenticated user name, if any.

Storing and replaying cookies using `curl`. The `curl` tool allows recording and presenting cookies through its “cookie-jar” flags, `-c` and `-b`. Type the command

```
curl -d "username=bezos&password=amazon" \
  -c cookies.txt -b cookies.txt
http://127.0.0.1:45006/
```

and see the `cookie token` stored in the file `cookies.txt`.

Subsequent browser requests should contain the cookie. If you open a new browser tab, pointing it to `http://localhost:45006/`, you must be able to see the `Cookie:` header printed in the request headers printed on `server.py`'s output in the terminal!

Step 6: Implement cookie-based authentication

Finally, we are ready to implement cookie-based authentication for users accessing our server. Browsers, and in general any HTTP clients implementing cookies correctly (including `curl` with cookie jar), are expected to present any cookies provided by a server in subsequent HTTP requests to that server.

Your task now is to validate these cookies on the server side. From the HTTP request headers, extract the `Cookie:` header, and check if it corresponds to any user whose cookie you already recorded in step 5.2.

Case C. Cookie validated. If there is a `Cookie` header in the request, and the cookie is one of the pre-recorded cookies, this corresponds (with high likelihood) to a successfully authenticated user. (In this project, we will not consider “attacks” where cookies are stolen from a user and presented on their behalf or a malicious client checks all possible cookie values by brute force.) In this case, the output is similar to that of case A in step 4: welcome the user and show their secret content.

Case D. Cookie invalid. If there is a `Cookie` header in the request, and the cookie's value isn't one of the pre-recorded ones, you must present an error to the user similar to case B from step 4.

Case E. No cookie header. If there is no `Cookie` header in the request, you must process the request as if you're doing Step 4.

Summary of application logic. After completing this step, the flow of application logic should look like:

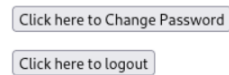
1. (case C) cookie header present and valid. Show the user the secret page: send back `success_page` with the secret.
2. (case D) cookie header present but invalid cookie: send back `bad_creds_page`
3. (case A from step 4) cookie header absent, username-password present in entity body, user-pass combination successfully matches correct credentials from passwords file: send back `success_page` with the secret, *don't forget to set a new cookie!*
4. (case B from step 4) cookie header absent, either one of username/password fields missing or username-password both present in entity body, but user-pass combination does not match any correct credential from the passwords file: send back `bad_creds_page`
5. (basic case, already implemented in the starter code from step 1) cookie header absent, username-password absent in entity body. Send back `login_page`

Step 7: Implement change password function

We get to change passwords for users accessing our server. Most websites allow users to change their password. Your task is to check for changed password and update the database with a new password.

Case F. Change Password: On successful login, you will see the `Click to change password` button on the Welcome HTML page.

Welcome!



Click here to Change Password

Click here to logout

Your secret data is here:

kaching

On clicking this button, you should be directed to new password page as shown below:



New Password:

Submit

To implement a change password function, all you need to do on the server side is to detect `password=new` in the POST entity body, and send `new_password_page` when that happens.

On entering the password and clicking the `submit` button on the new password page, you should be redirected to the success page. To implement this, detect `NewPassword` in the POST entity body. Retrieve the password associated with the `NewPassword` attribute and update the password in the database. Finally, return the success page.

Extra credit: Step 8: Implement logout function

Your browser will keep presenting the same cookie indefinitely if cookies aren't cleared or don't have an expiry date. The cookie we sent back in step 5 does not have an expiry date. Hence, a user, once logged in, must manually clear cookies each time they want to log out and avoid secret data being served by `server.py`. (Cookies can usually be cleared on a per-domain basis through the browser's cookie settings.)

In this extra credit step, you will send back a cookie header from the server that explicitly corresponds to clearing the cookie from the client. This is accomplished by setting an expiry date for the cookie that is in the past. For example, if you set `headers_to_send` to the string

```
Set-Cookie: token=; expires=Thu, 01 Jan 1970 00:00:00 GMT\r\n
```

it instructs the browser (HTTP client) to clear its cookie named `token` for the server's domain, since the cookie is past its expiry date. If your server sends this header in the response, you can check on your browser's cookie list as well as `curl`'s cookie jar file (`cookies.txt` in the

examples above) that the cookie has disappeared. (I have sometimes observed that `cookies.txt` itself disappears if there are no other cookies stored in it.)

Case G. Logout. Helpfully, the welcome HTML page that appears when a user is successfully authenticated contains a logout action button, that POSTs a hidden object called `action` with value `logout` to the server when the button is clicked. To implement a log-out function (clearing cookies in the process), all you need to do on the server side is to detect `action=logout` in the POST entity body, and send an expired cookie to the client when that happens. Further, send the HTML content corresponding to `logout_page` back to the user indicating they have logged out.

If you implement this extra credit step, the flow of application logic is enhanced by implementing step 8 at the beginning of the overall logic flow described in the summary at the end of step 6. That is, the full order of actions is

1. case F (step 8)
2. case C (step 6)
3. case D (step 6)
4. case A (step 4)
5. case B (step 4)
6. basic case (already implemented in step 1)

What you must submit and how we will test it

For your project submission on Canvas, please turn in `server.py` and your project report `report.pdf`. The command line of the program must be exactly the same as it was given to you (same as in step 1). The questions for the report are listed below. We will be running your code on the `ilab` machines with the default Python 2 version on those machines. **Please compress the files into a single Zip archive before uploading to Canvas.** Only one team member must submit.

Testing your program

We will test your program at least with the following cases and more. We may run the client on the same or a different machine from the one running your `server.py`. You may use this as a guide to check that your program implements all the required functionality correctly (we also mention the desired end result in parentheses). In all tests, all input files will be well formed, and we will only send well-formed HTTP packets crafted through a browser or `curl`. You should be able to craft requests corresponding to these cases through either your browser, or more likely, `curl`.

1. basic: no username or password posted, no cookies (login)
2. correct username and password posted, no cookies (success)
3. non-existent username posted with password, no cookies (bad credentials)
4. existing username posted with bad password, no cookies (bad credentials)
5. exactly one of username or password posted, other field missing, no cookies (bad credentials)
6. no username or password posted, valid cookie (success)
7. non-existent username or bad password for existing username, valid cookie (success)
8. correct username and password, valid cookie (success)

9. correct username and password, invalid cookie (bad credentials)
10. change password, no cookies (new password, then success)
11. (if you implement the extra credit) logout posted, valid cookie (logout)
12. (if you implement the extra credit) logout posted, invalid cookie (logout)

Project report

Please answer the following questions for the project report.

1. Team details: Clearly state the names and netids of your team members (there are 2 of you).
 2. Collaboration: Who did you collaborate with on this project? What resources and references did you consult? Please also specify on what aspect of the project you collaborated or consulted.
 3. Is there any portion of your code that does not work as required in the description above? Please explain.
 4. Did you encounter any difficulties? If so, explain.
 5. Describe **two** observations or facts you learned about HTTP and cookies in the process of working on this project. Please be specific and technical in your response.
- Contact the course staff on Piazza if you have any questions.