CS 161
Computer Security

Project 3 Part 2

Due: May 3, 2020

Most recent update: April 22, 2020

In the second part of this project, you will design and implement a secure version of the vulnerable website from part 1. This part of the project can be done with *one* partner. This project will not be as intensive as project 2–a secure implementation can be written in about 100-200 lines of code, depending on how you structure your code.

In order to aid in immersion, this project has a story. It is just for fun and contains no relevant information about the project.

> All hope seems lost as the CEO of UnicornBox proudly rings the opening bell of Caltopian Stock Exchange. Yet, in the soda-filled halls of Berkeley University, relentless students complete the final lines of debilitating proof-of-concept exploits against UnicornBox. The exploits, hot off the keyboard, are mailed straight to The Daily Caltopia.
>
> Students wait in anticipation, eyes glued to the TV screen. One moment, two people are bantering in a talk show. The next, a news anchor stutters as he reads off a slip of paper. 'Breaking news', the headline reads. 'University of Berkeley uncovers crippling security flaw in UnicornBox software.'
>
> UnicornBox shares plunge in shambles as users scramble to get sensitive data out of the servers. Prosecutors are forced to drop all charges against EvanBot, its claim proven true by valiant efforts of Berkeley University students.
>
> In the span of the next two hours an emergency board meeting ousts the CEO and, in an unexpected turn of events, invites EvanBot as the new CEO!
>
> As EvanBot makes its way from the court to UnicornBox board meeting, determined to bring down the evil conspiracy for good, it catches a glipse of a roadside billboard advertising UnicornBox. 'Where is your dream?' the billboard reads, along with other messages graffitied in by disappointed vandals.
>
> At that very moment, EvanBot realizes the current disappointment for UnicornBox is only made possible by the great hype that preceded it. People *are* yearning for a breakthrough in fileshare. People *want* what UnicornBox promised to deliver.
>
> So right there and then, EvanBot decides against shuttering UnicornBox. EvanBot now hopes to make good on the promise. To bring the true innovation Caltopia deserves, security and privacy included.
>
> And so EvanBot once again asks for help from its true friends.

Getting started

Your task is to fill in six missing functions in the UnicornBox server implementation. For simplicity, the UnicornBox server in this part does not implement the rename file feature.

To get started, pull the starter code from the following repository:

https://github.com/cs161-staff/project3-starter-code.git

Make sure you are using Go version `>=1.13`. You can check this by running `go version`.

Additional notes

- You may import any Go standard libraries that you may find helpful. However, you may not import any third-party Go libraries, except for the provided logging library `github.com/sirupsen/logrus`, or else the autograder will complain.
- EvanBot does not recommend changing any code outside of the predefined `YOUR CODE HERE` sections. In particular, the autograder will complain if you change the default server port or any of the HTML files in the `template` folder. If you must change the starter code, EvanBot recommends making a private post on Piazza before making any changes.

Autograder

- Like project 2, we will be releasing some limited sanity tests before the project due date. Your submission will be evaluated against all our test cases after the project is due.
- The autograder compiles your code with Go version 1.13, starts the web server locally, and makes HTTP calls to the server. After each run, the database and local disk storage are cleared.
- Unless otherwise specified, the functions you implement should return HTTP responses. The autograder will check the status code and, if relevant, the content of the response you return.
  - If the function call is successful, the HTTP response should have status code 200 and the correct content.
  - If the function call fails, the HTTP response should have any non-200 status code. The autograder will not check the content of the response if the function call fails, so you can put anything in the content to help with debugging.

UnicornBox behind-the-scenes

You will implement the webserver in Go. Go provides a framework for implementing web applications. Whenever a user visits a page on the website (such as `https://yoursite.com/list`), the framework will call a function you write to handle such requests (e.g., `listFiles`). This function is passed information about the HTTP request, and is responsible for assembling a response to be sent to the user.

The response contains a HTTP status code (200 for success, something else for an error) and, if applicable, the HTML contents of the page to be sent to the user. We use templates for the HTML, which consist of static content and some holes that can be filled in at runtime by your function.

We also provide a SQLite database and disk storage for files. If you need to store information that will be retained after the request completes, it should be stored in the database or on disk. Do not store persistent information in global variables or data structures in memory—instead, store it in the database or on disk, so that if the webserver gets restarted everything will continue to work. This is a standard way of structuring modern web applications.

Code walkthrough

To familiarize you with how to implement functionality in this framework, EvanBot has prepared a walkthrough of the login code (already implemented for you) as a Google Slides presentation. We recommend reading at least up to slide 21 before writing any code. The list of relevant functions is also available in an appendix at the end of this spec for your reference.

As you work on the project, you can use this walkthrough to help you understand what each line of code is doing and look up relevant syntax.

Starting the server

To run the server locally, execute `go run *.go` in terminal and navigate to
`http://localhost:8080/` in a browser.

Resetting your database and disk

If you want to clear the database and disk, delete everything in the `files` directory and the `test.db` file.

If you update the `files` table schema in `database.go`, remember to clear the database by deleting `test.db` so that your changes can be applied.

1. User authentication

Complete the `UserAuth` helper function in `middlewares.go`.

This helper function will be called on every request, to try to identify the currently-logged-in user that should be associated with this request (if possible). You should do that by:

- Extracting the session token from the cookie in the HTTP request (this is done for you)
- Look up the session token in the `sessions` table
- Make sure the session token exists in the database, i.e. your query returned a value
- Check the expiration time to ensure that the session token has not expired
- If the token is valid, assign the corresponding username to the `username` variable and run the following line of code:

  ```
  request = request.WithContext(context.WithValue(request.Context(), userKey,
  username))
  ```

- Before returning, run the following line of code: `next.ServeHTTP(w, request)`

Note: The original version of the starter code incorrectly said that you only need to run `next.ServeHTTP(w, request)` if the session token is invalid. Make sure that you run `next.ServeHTTP(w, request)` before returning even if the session token is valid.

Relevant walkthrough slides: 1-21 (to familiarize yourself with the starter code)

2. Logout

Implement support for logging out by completing the `processLogout` function in `controller.go`.

Your code should clear the session token cookie in the user's browser and delete the session from the server's database.

Relevant walkthrough slides: 22 (setting cookies)

The files module

The rest of this project involves completing the file functionality of the server. A correct file server implementation must meet the following requirements:

Security

- Each file should only be accessible by its owner and users who the owner has shared it with.
- Only the owner of a file (the user who initially uploaded it) should be able to share it with others.
- Your server must defend against XSS, SQL injection, and other standard attacks.

Functionality

- Different people must be able to upload different files with the same name.
- We will not test the following scenarios:
  - A user uploads a file with the same name as a file they have already uploaded or a file that has been shared with them.
  - A file owner shares a file with the same recipient twice.
  - A file owner shares a file with a recipient who already has a file with the same name.
- File owners should not be able to share files with themselves or with nonexistent users.

Filenames

- Filenames may only contain a-z, A-Z, 0-9 and `.` (the period). All other characters are considered invalid.
- Filenames must be between 1 to 50 characters long (inclusive).
- Any attempt to access an invalid filename should result in a non-200 HTTP code being returned.

Storing files and filenames

For the rest of this project, you should update the schema for the `files` table in `database.go` as you see fit.

If you update the `files` table schema in `database.go`, remember to clear the database by deleting `test.db` so that your changes can be applied.

We recommend storing uploaded files in the directory `const filePath = "./files"`.

3. Upload files

Add support for uploading files by completing the `processUpload` function in `controller.go`.

Relevant walkthrough slides: 23-27 (reading and writing files)

Design tips: What will you store in the database and what you will you store on the filesystem? What information needs to be stored? Think in advance about how you'll support Tasks 4–6 below, and what information you'll need to save to make them possible. What strategy will you use to defend against SQL injection?

4. List files

Add support so users can list their files by completing the `listFiles` function in `controller.go`.

This function has been partially implemented for you. Your code should fill the `files` slice with one `fileInfo` struct for each of the user's files. The starter code will take care of producing the output HTML page.

5. Download files

Add support for users to download files by completing the `getFile` function in `controller.go`.

You do not need to protect against malicious file contents.

Relevant walkthrough slides: 28-29 (attaching files in the response)

Design tips: How will you ensure that only authorized users can download the file?

6. Share files

Add support for users to share files they have uploaded with others, by completing the `processShare` function in `controller.go`.

Appendix: Relevant function documentation

The functions shown in the walkthrough slides are also described in the Go standard library documentation, for your reference as you work on the project. Relevant slide numbers are included in parentheses.

Database

`db.QueryRow`: Fetch one row from the SQL database. (9)

`db.Exec`: Execute a SQL query without returning any rows. (17)

`db.Query`: Execute a SQL query, returning multiple rows that can be iterated over. (18)

`row.Scan`: Read a row returned by a SQL query into variables. (11)

Writing to the response

`response.WriteHeader`: Writes a non-200 error code to the response. (13)

`fmt.Fprintf`: Writes debugging information to the response. (15)

`http.ServeFile`: Attaches a file to the HTTP response. (28)

Reading HTTP requests

`request.FormValue`: Extract a value in a form sent through an HTTP request. (8)

`request.FormFile`: Extract a file uploaded and sent through an HTTP request. (23)

Reading and writing to disk

`ioutil.ReadAll`: Extract the contents from a file object. (24)

`ioutil.WriteFile`: Write a file to disk. (25)

Cookies

`http.Cookie`: A struct representing a cookie returned in an HTTP response or sent with an HTTP request. (22)

`http.SetCookie`: Sets a cookie in the HTTP response. (22)