Application Security HW 2
Write Up
Harish Saravanakumar (hs3209)

GitHub Link: https://github.com/HarishSaravanakumar/AppSecHw2

The assignment can be represented in three main parts. Additionally, I will discuss the security aspects I added after creating the base application.

1. Turn the spell checking system from into a Web service using Flask
2. Create tests, and run them in pytest
3. Add more security to prevent various common web attacks (XSS, CSRF, session hijacking, etc)
4. Pass gradescope tests

My spell checking system consists of 2 main parts:

1. app.py with functions:
    a. home: check current login status; logout of current session
    b. register: create an account
    c. Login: log in to account
    d. spellcheck: check the spelling of text using the spell check binary created in assignment 1 and return the misspelled words
2. Corresponding .html(files: _formhelpers, home, register, login, spellcheck, spellcheck_results): These hold the messages to be displayed

I will also discuss basic security measures I added while creating this application.

Preliminary Steps:

I first created the Flask application. I configured a sqlite database to hold a user's credentials by creating a *database_uri* for the database to reside in. I also created a table (*userCreds*) to store the user's credentials, which includes the username, password, and two-factor authentication. The username flag is set to unique. I reset the tables when the flask application is launched so the database does not contain user credentials from the previous run. Following this, I defined two forms, using WTF-Forms and its method of form validation. I created templates in the form of classes: one for registering/logging in (*registerForm***)** and another for spell check (*spellForm***)**. In *registerForm*, I forced the length of the two-factor input to be 11 and gave it a corresponding **id='2fa'**. For *spellForm*, I created a *TextAreaField* for the user to input their text.

Endpoints:

1. **Home:** The home endpoint supports GET and POST requests. The endpoint exists to show the user whether they are logged in or not and to allow the user to log out. When the user registers and then logs in to their account, their login is represented in the

session dictionary. As a result, we can check if the user is signed in using *session.get('bool_log')*. If this is true, we can show the message *'Logged In'*, if not, we show the message *'Please Login'*. When the user presses the logout button, we need to pop their session from the dictionary and show the message *'Logged Out'*. Depending on which action occurs, **home.html** is rendered in with a different message.

   a. **Home.html:** The html has links to all other html files. Returns either "Logged In", "Please Login", or "Logged Out", depending on the action.

2. **Register:** The register endpoint supports GET and POST requests. The endpoint exists to allow the user to register for an account. We use the registerForm class we previously created to hold the credentials inputted from the user, then later pass it to the *userCreds* table. We first need to check whether the username already exists within *userCreds*, since we need to make sure the usernames used are unique. If not, we return the message *"failure"*. If so, we return the message *"success"* and add the inputted username, password, and two factor to *userCreds*. Depending on which action occurs, **register.html** is rendered in with a different message.
   a. **Register.html:** The html has links to all other html files. The register.html renders the *registerForm* for the user to fill out. Returns either "success" or "failure", or "", depending on the action.
3. **Login:** The login endpoint supports GET and POST requests. The endpoint exists to allow the user to login to their account. We use the registerForm class we previously created to hold the credentials inputted from the user. We check whether the user is already logged in using *session.get('bool_log')*. If so, we return the message "Already logged in". We then check whether the username exists in *userCreds*. If not, we return the message *"Incorrect"*. If so, check whether the passwords and two factor match the ones in *userCreds*. If so, return the message "success" and log the user in by changing the session status of the user to *True*. If not, we need to find what credential is incorrect. If the password is incorrect, return the message *"Incorrect"*. If the two factor is incorrect, return the message *"Two-factor failure"*. Depending on which action occurs, **login.html** is rendered in with a different message.
   a. **Login.html:** The html has links to all other html files. The login.html renders the *registerForm* for the user to fill out. Returns either "Incorrect", "Two-factor failure" or "success", depending on the action.

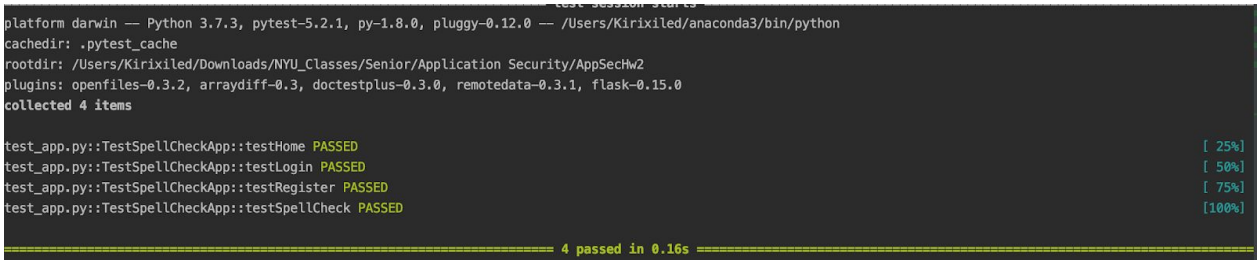
4. **Spellcheck:** The spellcheck endpoint supports GET and POST requests. When loading the spellcheck endpoint, we want to return the message *"inputtext"*. When the user presses the button to check spelling, we can use the spellForm class we previously created to hold the text inputted by the user. First, however, we need to make sure the user is logged in by using *session.get('bool_log')* again. If not, we return the message *"Please log in first!"*. We then create a text file *temp.txt* and write the data from the form into it. Here, we create a subprocess, opening the spell check application (*a.out*) and passing both the text to test (*temp.txt*) and the dictionary to use (*wordlist.txt*). After waiting for the process to complete, we return its output (the list of misspelled words) Since this output has not been parsed, we parse the output by replacing new line characters with commas. If this works, we return the data and output to *spellcheck_results.html*. We return the message *"success"*. If there is an error when using/calling the subprocess, we print out the error to the terminal, and return the message *"failure"*.
    a. **Spellcheck.html:** The html has links to all other html files. The spellcheck.html renders the *spellForm* for the user to fill out. Returns "inputtext", "Please log in first!", "extraneous error", depending on the action.
    b. **Spellcheck_results.html:** The html has links to all other html files. The spellcheck_results.html only returns the supplied text and misspelled words from the spell check subprocess.

_formhelpers.html: Used in order to allow html to use render_field

Basic Security: I added exceptions for my functions and returned an output for every case. Also, in order to display output safely through my html files and prevent cross-site scripting (XSS), two things are done. Firstly, Flask uses Jinja2 to automatically escape all values unless explicitly told otherwise. In addition to this, to prevent XSS by attribute injection, I surrounded my form fields with quotes.

Testing:

I created a couple of tests and tested them in pytest rather than tox. The code passes all of them.



They check whether each endpoint is working correctly (OK status).

Finding Vulnerabilities

After reading the various links and taking an introspective look at my own code, I found several problems with it.

1. When creating the subprocess, I passed arguments (*/a.out*, *temp.txt, wordlist.txt*) directly into the subprocess i.e: **popen = subprocess.Popen("/a.out", "temp.txt", "wordlist.txt", stdout=subprocess.PIPE)**. This is problematic because there is no separation between the input parameters. As a result, the shell cannot tell where the command should end, and the parameters (like stdout) begin to be defined.
    a. Solved: I pulled out the arguments and stored them in a list. I then passed this list into the subprocess.
2. The subprocess was not terminated once the output was stored into a variable.
    a. Solved: Terminated the subprocess.
3. Passwords were not hashed in any way. As a result, attackers can easily pull passwords.
    a. Solved: I hashed the passwords using bcrypt. First, I passed the app to bcrypt. From here I had access to two methods, *generate_password_hash* and *check_password_hash*. The first method was used when a user registers (**register** endpoint) for an account. We pass the password the user inputted to *generate_password_hash*, which hashes the password. Instead of storing the password the user passed in to *userCreds*, we store the hashed password. This way, even if the attackers were able to get a hold of the hashed password, they

would not be able to decode it in order to log in to the user's account. When the user logs into their account, the inputted password is compared to the hashed password in *userCreds* using *check_password_hash*.

4. Did not protect against CSRF (Cross-Site Request Forgery) attacks.
    a. Solved: Used CSRF Protect from Flask-WTF in order to protect against attacks. First, I passed the app to csrf. Since csrf needs a secret key, I created one using *os.urandom*. Since my templates do not use *FlaskForm*, I hid each input in my htmls (ex. <input type="hidden" name="csrf_token" value="{{ csrf_token() }}"/>).
5. Did not protect against session hijacking attacks. Although sessions were used, they were not encrypted in any way. As a result, attackers can easily impersonate an identity, and use the spell check application without logging in.
    a. Solved: Created a secret key for the sessions to use. Since I had already created a secret key for CSRF, I reused that key for securing my sessions.

Finally, I passed all of the tests on Gradescope.