Nicolas Gomez
Professor: Kevin Gallagher
Application Security
October 20, 2019

# Unit 2

As a continuation of Unit 1, I moved on to learning about Web Development, starting a web service using Flask, and how to implement security on web applications. To start, I followed the instructions on the Assignment guidelines and installed Pip3, Flask, and Tox. Pip is a package management system that is used to install and manage software written in Python. Flask is one of the most popular lightweight web application frameworks. Tox is a tool utilized to automate testing in Python, in this case, I will use Tox to test the web service created via Flask.

The service that needs to be created, will need to allow users to register, login, logout, and be able to use the spell check tool only when the user is logged in. In order to do this, there is a need for a database to store the user's credentials, some type of session keeping system, and the service should be able to handle different types of GET and POST requests.

## Week 4

For this week I worked on creating a web service with the help of Flask. To start, I created the "your/webroot" directory and inside that directory I created a file named "app.py". With the help of the Flask documentation, I wrote a simple application that returns "Hello" when you visit "localhost:5000/" after running the "env FLASK_APP=app.py flask run" to start the service. I then created five route decorators to tell Flask what URL triggers which function.

- "/" route:
    - For this function, I set up a redirect to go to the /login URL.
- "/register" route:
    - The function allows the user to register their accounts into the service after they enter their credentials. This will allow them to login to use the spell check tool. This function accepts both GET and POST requests.
    - GET:
        - Returns the rendered template of the register.html web page which has a form that has three inputs and a submit button.
            - Input 1 is the username field that has ID uname
            - Input 2 is the password field that has ID pword
            - Input 3 is the two-fa factor field that has ID 2fa
        - After the user enters their credentials a POST message is sent with the information to the service.

    - POST:
        - If the service receives a POST message on this route, it will parse the inputs from the user and add them to the database. Before it can save them, it will check if the username is already in use by checking the database. If not, it will add the credentials to the database and render the page with a message of "success" in a field with ID success. If the username is already in the database, it will return a message of "failure" in the field with ID success.
    - The database used in week 4 was stored in a plain text file named database.txt where everything was stored in plain text and passwords were not encrypted. The function would append to the file as information came in and read from it when trying to find if the username exists or when trying to login. This was fixed in Week 5.
- "/login" route:
    - The function allows the user to log in the person into the service after they enter their credentials. A session is started to allow them to use the spell check tool. This function accepts both GET and POST requests.
    - GET:
        - Returns the rendered template of the login.html web page which has a form that has three inputs and a submit button.
            - Input 1 is the username field that has ID uname
            - Input 2 is the password field that has ID pword
            - Input 3 is the two-fa factor field that has ID 2fa
        - After the user enters their credentials a POST message is sent with the information to the service.
    - POST:
        - If the message received is POST, the function will parse out the data entered by the user. It then checks if the information is in the database to authenticate the user. If the username or password is incorrect, a message will be returned to the user in the rendered page in a field with ID result with an error message saying incorrect. If the username and password are correct but the two-factor authentication is not, the field with ID result will show an error message saying "Two-factor failure". If all the fields are correct for the user a session is started allowing the user to use the spell_check tool.
    - The session started would save the username of the person until they logged off or the service was shut off. This was a flaw that I fixed in week 5.
- "/spell_check" route:

    - This function allows users to enter bodies of text to check for spelling errors. This function accepts both GET and POST requests. The user will only be allowed to see the input box if there is a session present. If not, they get a message saying that they need to login. This is controlled by an if statement in the HTML page that pulls session information.
    - GET:
        - Returns the rendered template of the spell_check.html web page which has a form that has one input with ID inputtext and a submit button.
        - After the user enters their body of text into the box, a POST message is sent with the information to the service.
    - POST:
        - If the message received is POST, the function will take in the text entered by the user and write it to a temporary file. Once that file is saved, the file is checked with the spell_checker binary by running a subprocess and reading the standard output to see if any words were misspelled or not. If words were misspelled, they are shown to the user in a paragraph with ID misspelled. The text the user entered is also shown in a paragraph with ID textout.
- "/logout" route:
    - This function terminates the session of the user by using the pop() function against the session to remove the username from it.
- 404 errors route:
    - I was able to setup a route that handles 404 errors when a page is not found which returns a rendered page with links to the other route according to the session status.

After the service was working as intended, I moved on to writing a functional test to ensure the service continues to work when changes are made. I also updated the .travis.yml file to automatically run these tests. I created a file named test_flask.py where I wrote all the tests for this service.

- test_index(client)
    - Test that when a user goes to localhost:5000/ they are redirected to the login page. It checks if the response status is 302.
- test_register_page(client)
    - Tests that users are able to reach the localhost:5000/register page by checking for a 200 response from the service.
- test_register_entry_successful(client)
    - Tests that users are able to reach register an account by sending test credentials and checking for a 200 response code and the success output from the field with ID success.

    - After this, the test credentials are removed from the database.
- test_register_entry_already_exists(client)
    - Tests that users are able to not able to register an account when the username already exists by sending some test credentials with an existing username and checking for a 401 response code and the failure output from the field with ID success.
- test_login_page(client)
    - Tests that users are able to reach the localhost:5000/login page by checking for a 200 response from the service.
- test_login_valid(client)
    - Tests that users are able to log into the service by sending known test credentials and looking for a 200 OK response and a success message from a field with ID result.
- test_login_invalid(client)
    - Tests that users not able to log into the service by sending wrong username or password test credentials and looking for a 401 response and an incorrect message from a field with ID result.
- test_login_invalid_2fa(client)
    - Tests that users are not able to log into the service by sending the right username and password but the wrong 2fa test credentials and looking for a 401 response and a failure of 2fa factor  message from a field with ID result.
- test_spell_check_page_no_access(client)
    - Tests that the user does not have access to the page if they are not logged in by checking for a 401 response code.
- test_spell_check_page_access(client)
    - Tests that users are able to access the spell_check page by sending known test credentials to the login page and then going to the spell_check page and looking for a 200 OK response.
- test_spell_check_page_access_input(client)
    - Tests that users are able to access the spell_check page and send information to be spell checked by sending known test credentials to the login page, then going to the spell_check page, sending test text into the field with ID inputtext and checking for a 200 response and the test text being returned in the field with textout field and the misspelled word in the field with ID misspelled.

# Week 5

With the web service working as intended, we moved on to ensuring the site is protected by preventing against XSS, CSRF, Session Hijacking, and other web vulnerabilities. To start I looked into implementing a lot of the ideas from Oleg Agapov and its blog of https://codeburst.io/jwt-authorization-in-flask-c63c1acf4eeb like adding JSON Web Tokens (JWT) to have a better implementation of web security, SQLAlchemy for implementation of SQL database, and cookies for session keeping.

I removed the plain text database and replaced it with SQL Alchemy for better protection of credentials. In a file named models.py, I created a database with 4 columns, one with id which is a number, uname for the username which is a string, pword for the password which is a string, and 2fa for two-factor authentication, which is also a string. There are also some functions that I used from Oleg's code like find_by_username, save_to_db, generage_hash, and verify_hash. All of these functions help find and pull information from the database. I created a delete_user function that deletes the row with the information of the user to allow the deletion of credentials. The library passlib.hash was imported to support password and two-factor authentication encryption for better safe keeping of the passwords. Since the input is not using the execute function to pull information, it protects it from SQL injection attacks. Since the functions uses the filter_by function it protects it by taking information literally and only concentrating on the column specified. I updated the login and registration routes in the app.py file as well as the tests to support the SQL database.

With the SQL database setup and protecting users password by hashing them with SHA-256, I moved onto adding JWT support into the service and adding cookies support for proper session keeping. To do this, I followed the documentation on https://flask-jwt-extended.readthedocs.io/en/latest/tokens_in_cookies.html to learn how to set cookies and save tokens into them. To start I imported the flask_jwt_extended library and added some config values to the app as shown below.

- JWT_TOKEN_LOCATION
    - Cookies
        - Whenever a request is made, the user will need to send an access token via cookie. JWT creates a httponly cookie to protect against XSS attacks by preventing it to be access via javascript. This cookie replaced the session library used in week 4.
- JWT_ACCESS_COOKIE_PATH
    - /spell_check
        - This ensures that users need a valid cookie to access the /spell_check site and use it.
- JWT_COOKIE_CSRF_PROTECT

    - True
        - This enables CSRF (Cross Site Request Forgery) double submit of protection by requiring the sending of an CSRF token with POST requests that is added to the headers of the response.
- JWT_SECRET_KEY
    - ‘Something-secret’
        - This is a secret key to sign the JWTs with.

On the login route, I added the calls to the create_access_token and create_refresh_token functions and passing the identity of the user to generate the cookies and then setting the access cookies with the set_access_token and set_refresh_cookies functions. In the spell_check route, I added the @jwt_required decorator to ensure that only users with a valid cookie are able to access this protected page. When rendering the spell_check page, a hidden value is added to the form containing a csrf_token to enable the double submit protection which is sent back to the service with any POST request for protection against CSRF attacks. To end, I added the following headers to every route for further protection.

- Content-Security-Policy
    - default-src self
    - This header helps protect against vulnerabilities of XSS and clickjacking by not allowing any external resources on the site.
- X-Content-Type-Options
    - nosniff
    - Prevents the MIME Type sniffing functions in web browsers.
- X-Frame-Options
    - SAMEORIGIN
    - Prevents Clickjacking attacks by preventing outside pages in iframe or frame renders in the page.
- X-XSS-Protection
    - 1; mode=block
    - Prevents XSS by blocking XSS payloads from being loaded to the page.

Note: The addition of cookies and the CSRF protection broke my test test_spell_check_page_access_input which ensures users are able to submit text properly. I am still working on a fix to this issue.

# Week 6

In week 6, I worked on testing the application even further by running it against open source web vulnerability scanners that are pre-installed in Kali Linux and others that I found online. The first one that I wanted to test was the Arachni Scanner which is a web scanner built on Ruby. I downloaded the application and tried to scan the web pages but for some reason the tool is not able to scan items the Loopback interfaces. This is currently not supported by the tool as some users have reported in https://github.com/Arachni/arachni/issues/699. I also tried to change the IP where the service run to 127.0.0.2 and it did not work.

I moved on to using Metasploit and the WMap plugin which allows the scanning of web applications within the Metasploit Framework. I followed the instructions in their documentation page and ran the command "wmap_sites -a http://127.0.01:5000/" and "wmap_targets -t http://127.0.01:5000/" to add the site as a target, then running "wmap_run -e" to start the WMap scan of the service. Once it finished I saved the output to a text file named wmap.txt and noticed that Metasploit had not found any vulnerabilities on the site after running tests like SQL injections, directory traversal attacks, and host header injections.

Since I am using a SQL database, I decided to test the service with SQLMap. I ran the command "sqlmap --output-dir=/root/Desktop/Application-Security/your/webroot/vuln-scan/ --level=5 --risk=3 --data=uname -a --dump-all -u http://127.0.0.1:5000/register" which scans the http://127.0.0.1:5000/register site which takes in user input where one of the inputs has field id of uname, a level of test of 5, risk of 3, and tries to retrieve everything as described with the -a flag and dump all DBMS databases table entries. All of the output was saved to the location specified in the --output-dir flag, a file containing the output of the command as it ran was saved in the sqlmap.txt file. The output of the tool showed that the site was not injectable after multiple SQL injection attacks were run against the service.

Nikto is another open-source tool that aims to find web server misconfigurations, vulnerabilities, and plugins. I followed the Nikto tutorial and ran the command "perl nikto.pl -host http:127.0.0.1:5000/" and it gave me the output which was saved in nikto.txt file stating that the security headers of X-Frame-Options, X-XSS-Protection, and X-Content-Type-Options are not set. I found this odd since I specified these headers in every response made. Note: I am still working on ensuring all security headers are working as intended. Apart from that, the Nikto tool did not find any other vulnerabilities or issues.

I also tried using other open source tools like w3af, wfuzz, and wapiti but due various issues like the program not building properly, missing dependencies I was not able to install, and other bugs I was not able to figure out, I could not use these tools. With some manual testing I made sure attacks like XSS are not occuring by entering the javascript "<script> alert("Hello! I am an alert box!");</script>" in the register page and checked if any javascript was executed, which it didn't. A part of the service is to put the username of the user in the page to show how it

is the user logged in, meaning if I saved this javascript as a username for an account, the javascript might be executed when the username is rendered. I tested this and it did not work, when checking the code of the page I noticed the symbols had been encoded as the following “<script>  alert("Hello! I am an alert box!"); </script>”.  I did the same with SQL injections and every command I tried to pass, it would get saved into the database and when I tried to log into it, it would work fine and the output was encoded,  no extra information was returned to the user.

# Resources

- Assignment Github: https://github.com/ng1968/Application-Security/tree/master/your/webroot
- Assignment Guidelines: https://drive.google.com/file/d/1hEBvDuCOvQ05dxlsbYxDyv6g-RiMJkhX/view
- Arachni Scanner: https://www.arachni-scanner.com/
- Oleg Agapov - Codeburst: https://codeburst.io/jwt-authorization-in-flask-c63c1acf4eeb
- Flask Documentation: http://flask.palletsprojects.com/en/1.1.x/
- Flask JWT Extended: https://flask-jwt-extended.readthedocs.io/en/latest/tokens_in_cookies.html
- Flask Session: https://pythonhosted.org/Flask-Session/
- Metasploit WMAP: https://www.offensive-security.com/metasploit-unleashed/wmap-web-scanner/
- Models.py file: https://github.com/ng1968/Application-Security/blob/master/your/webroot/models.py
- Nitko: https://hackertarget.com/nikto-tutorial/
- Pip: https://pypi.org/project/pip/
- Security HTML Headers: https://www.netsparker.com/whitepaper-http-security-headers/#ContentSecurityPolicyHTTPHeader
- SQLMap: https://github.com/sqlmapproject/sqlmap/wiki/Usage
- Test_flask.py file: https://github.com/ng1968/Application-Security/blob/master/your/webroot/test_flask.py
- Tox: https://tox.readthedocs.io/en/latest/
- .travis.yml file: https://github.com/ng1968/Application-Security/blob/master/.travis.yml
- Unit 1 report: https://github.com/ng1968/Application-Security/blob/master/gomez_nicolas_report1.pdf
- W3af: http://w3af.org/
- Wfuzz: https://tools.kali.org/web-applications/wfuzz
- Wapiti: http://wapiti.sourceforge.net/