**Michael Tse**

**10/22/19**

**Application Security – Assignment 2 – Fall 2019**

**Github Repository -** <https://github.com/mt1836/assignment1_week2>

**INTRODUCTION**

In this assignment we were asked to take the spell checker created in Assignment 1 and turn it into a web application. This web app would be run on localhost:5000 and would store all necessary data in memory. There is no database in scope for this assignment. Once the web app is functioning, we create unit tests for functionality and identify security vulnerabilities and how they can be addressed.

**WEB APPLICATION CREATION**

The web app was created using a combination of python for the logic, flask for the web application framework, wtf for form management and jinja for html templating. The following three html pages were created to support the web app:

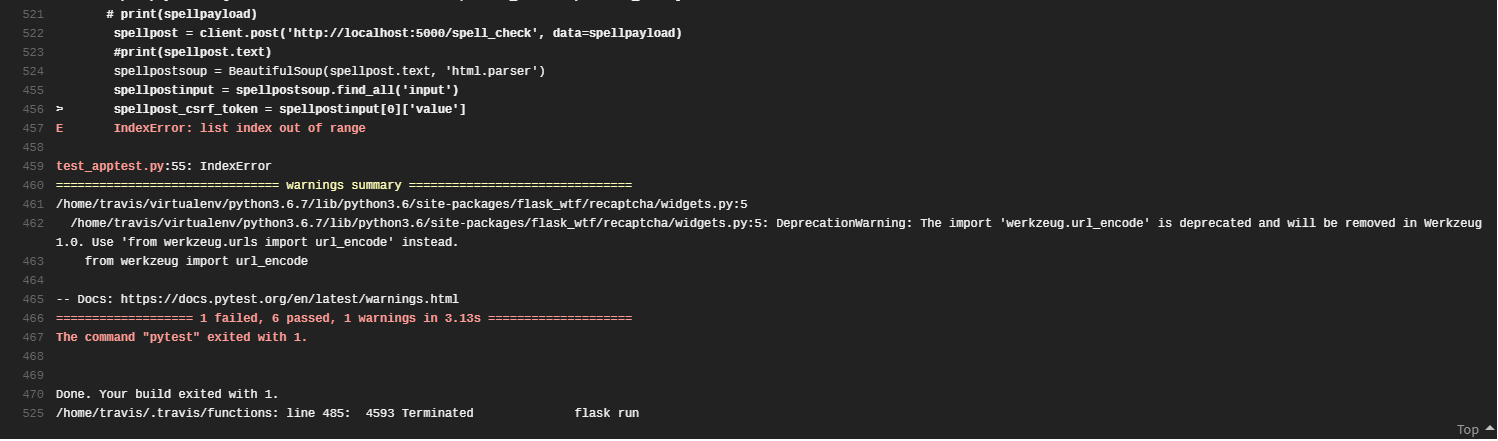
* Register.html: registers new users for the spell check service using 3 fields (username – required, password – required, phone number – optional). Username/password requirements are between 1-20 characters with phone number as optional. If supplied this will be used to mimic two-factor authentication.
* Login.html: registered users are able to login via the login.html page to gain access to the spell check web application.
* Spell\_check.html: once access is granted via the login page, users are able to enter a string of text in a form field and submit the data to the web application. The web app stores the input in both a string and a new text file and reflects it to the user along with the words that are misspelled. The text file is used as input when the spell check binary is called by a method called subprocess.check\_output.

**UNIT TESTING**

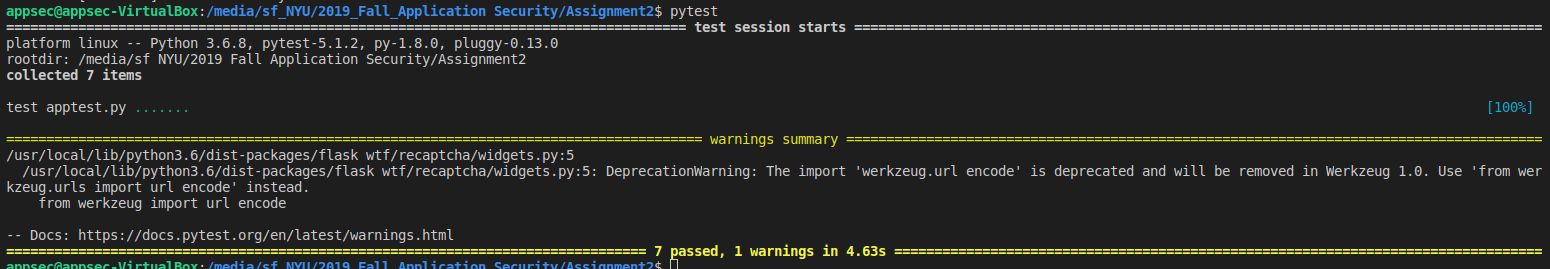
The goal of the build and testing is to utilize github as a repo for signed commits and automate the testing process via Travis CI and Tox. The unit tests were also tested locally using pytest which incorporates the use of requests to get and post data and beautiful soup to parse the html to make the necessary assertions. There are 7 unit tests created to test functionality of our web application:

1. Test\_pagesxist – includes three assertions to check for response code 200 for the register, login and spell check pages.
2. Test\_regsuccess – confirms that a new user can successfully register for an account by asserting that a success message is returned.
3. Test\_regfail – fails a registration if required fields are not completed. Asserts by checking that a failure message is returned.
4. Test\_duplicatereg – checks that registration will fail during attempts to register an existing username by asserting that the user exists message is returned.
5. Test\_invalidauth – checks that attempts to access the spell check page is denied if an invalid authorization (a user who was not registered) is received by asserting the “incorrect” message is received after a failed login attempt.
6. Test\_loginsuccess – checks that a registered user can successfully login by asserting the message success if returned.
7. Test\_spellcheck – first registers a user, then logs the user in and gets the spell\_check.html page. Submits text to be spell checked and asserted that the misspelled words returned are the ones expected.

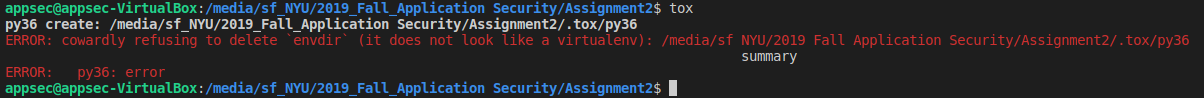
These 7 unit tests cover the functionality of the web application. Ideally we would have liked to run the unit tests through travis🡪tox which kick off pytest however there were complications with getting both Travis and Tox to function properly. The following error was obtained in our Travis output.



Running pytest locally we did not encounter any issues and all 7 tests passed.



When attempting to run tox, we were not able to identify the root cause of this error message:



Subsequently all unit tests were checked locally and not through Travis or Tox.

**SECURITY TESTING**

CSRF

Cross Site Request Forgery is an attack that forces an end user to execute unwanted actions on a web application that they are currently authenticated to. In the specific example of the spell checker, a successful CSRF attack would allow the attacker to initiate the spell checker with a string to be checked and return results back to the victim. In this scenario the CSRF would not have much impact as the attacker would not be able to see the results of the spell checker. While it is harmless in this application it demonstrates how it could work in a more important scenario such as a banking application where funds could be transferred to the attacker.

The CSRF attack would require that the victim be currently logged into the spell check application and be tricked into clicking on a link that has the proper URL encoding to submit form information to the spell checker using the victims open session. To defend against this attack CSRF tokens were implemented in our application. Upon every form request (get), a CSRF token is randomly generated by the web service and sent to the user in the get request located inside the <form> tag. If this CSRF token is not presented in the post request when submitting form data, the web service will do nothing with the post request. We were able to confirm this by submitting post requests without the CSRF token and no data was posted. The CSRF tokens are enabled as hidden values in the form field of each html page along with a secret key defined in the python app.py itself.



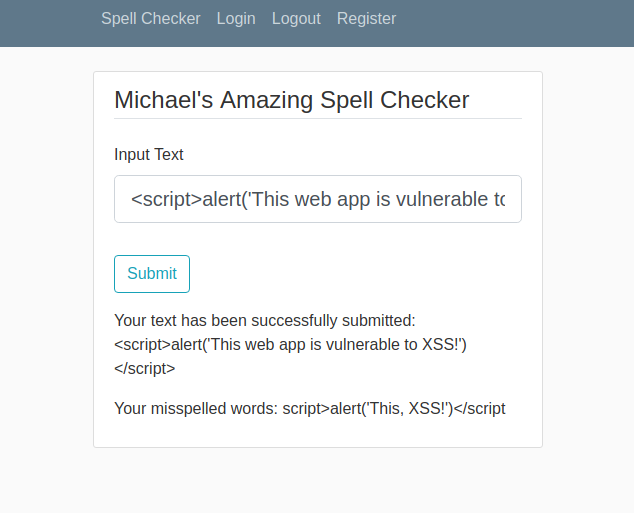


XSS

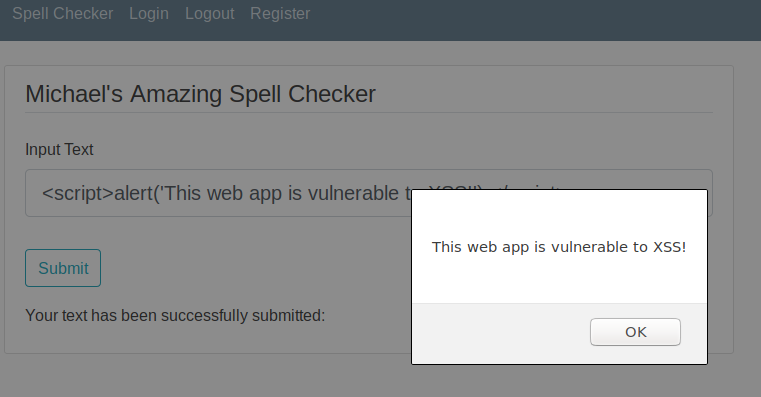
There are three types of cross site scripting attacks:

* Stored: Malicious script originates from a server and delivered to the victim. The code is stored in the server (database) by unsanitized input from form data that the attacker sends. When victims access the legitimate site and perform functionality that pulls the stored data, the malicious code is run on their browser.
* Reflected: Malicious script originates from a server and delivered to the victim. This type of XSS attack works when input data is reflected back to the user. In the example of our Spell Check app, the input text to be checked for spelling errors is reflected back to the user to show the original text. A malicious link could be sent to the victim and socially engineered to have them click on it sending them to the legitimate site. This link would populate form data with Javascript that could steal the victims session by requesting the victims browser to append cookie data and send back to the attacker.
* DOM: Javascript is run in the Document Object Model. The malicious script is found in the DOM of a dynamic site that changes according to user input (forums etc.) When subsequent users load the page, the html page will present the script to the user and force execution on the user’s browser. The script cannot be found by inspecting the HTML source code itself.

In this assignment we tested for reflected XSS by injecting the Javascript <script>alert(‘This web app is vulnerable to XSS!’);</script> which is expected to generate a pop up menu with the text “This web app is vulnerable to XSS!” should our spell\_check.html be vulnerable to XSS. When we initially ran this there was no pop up and the script was treated as a string and reflected as such per the screenshot below. This proves that our site is not subject to reflective XSS as input is being sanitized by WTForms. Other ways to thwart this would be to encode the output to ensure that characters such as ‘<’ and ‘>’ are treated as text.



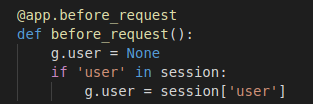
To prove that WTForms is in fact defending against this XSS attack, we modified our app.py to change {{input\_text}} in spell\_check.html to {{input\_text|safe}} which tells WTForms that the input data is safe and to accept it as is. With this modification we once again input Javascript <script>alert(‘This web app is vulnerable to XSS!’);</script> except this time we can confirm that the script ran successfully as the pop up window with the expected message was shown.



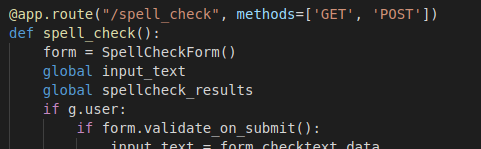
Because WTForms protects against reflected XSS thru the sanitization of input data it also protects against stored XSS. There are no objects in our DOM that would be subject to XSS. Additional protections against XSS can be performed through the implementation of Content Security Policies which allows script execution only from whitelisted sources.

Session

The spell\_check.html site was served to only successfully authenticated users be creating sessions for logged in users and requiring a user with an authenticated session before loading spell\_check.html. This was performed through the following code



and adding g.user to our spell\_check route.

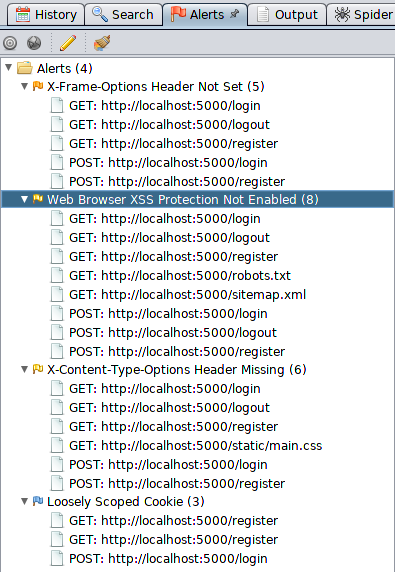


PasswordHashing

When collecting account information, passwords should always be salted and hashed before storing. Only the hashes should be use to authenticate. This protects against attacks which target the /etc/shadow file. Should the shadow file be obtained, the hash and salt will add a significant level of protection. We used bcrypt to generate the hash and salt



CSP (Content Security Policy)

We attacked our site using OWASP ZAP and received the alerts listed below. 

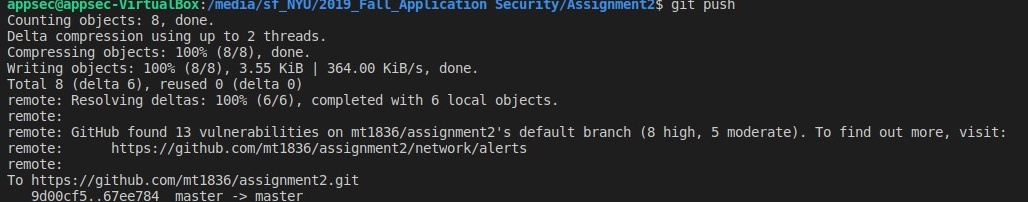
Some of the alerts are false positives as they pertain to CSRF but other alerts we felt would be addressed by implementation of a Content Security Policy such as the one below as a meta tag in the header of our layout.html:

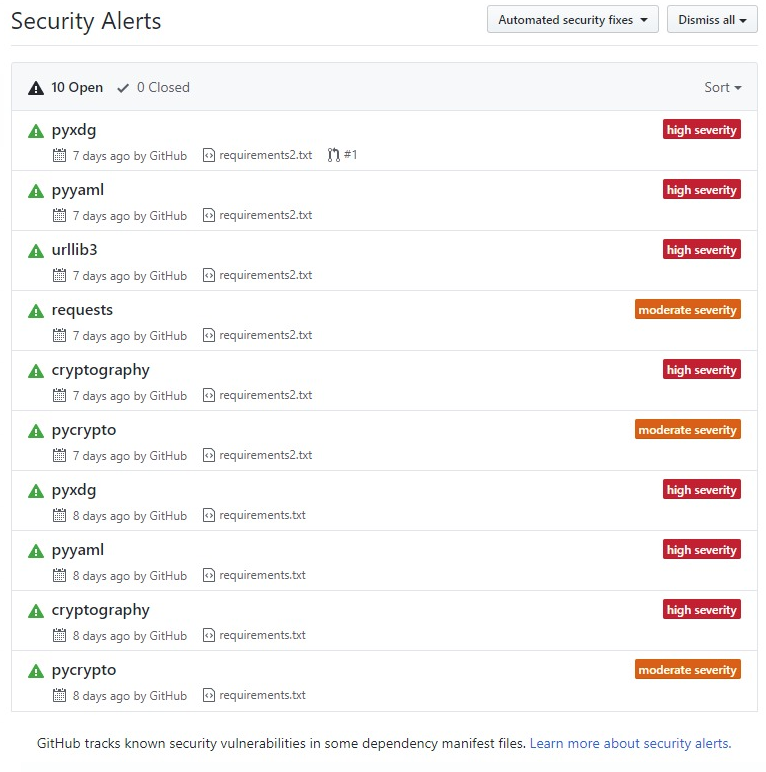


CSPs are a good catch all as it functions to whitelist approved activity on a site. There are many options that allow only activity from self or approved domains such as cookie-scope. We implemented default-src ‘none’ do disallow everything if there is no specific call out to whitelist. We then added script-src ‘self’ and child-src ‘self’ that allows us to run scripts and generate frames from within our own domain. Though this is not necessary in the spell check web app we implemented it to show how a CSP could be configured. There are many other options/configurations that one can add to a CSP but it is good practice to deny all and only provide functionality that is necessary.

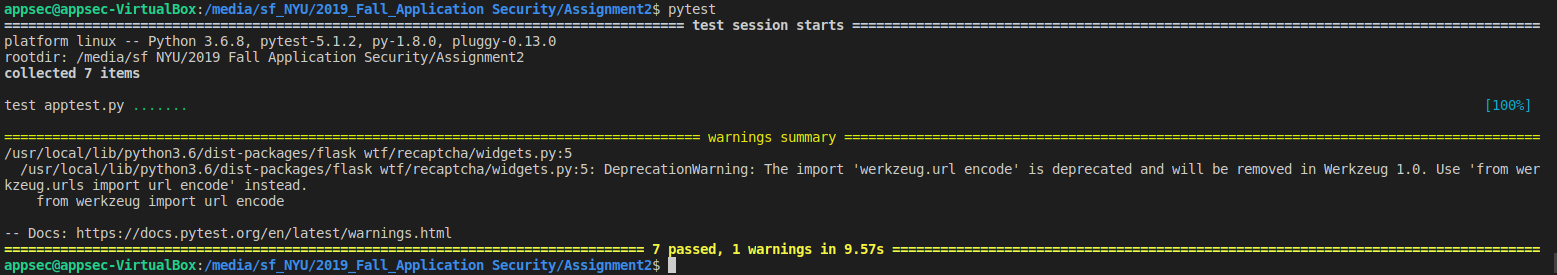
Miscellaneous

When pushing our code into github, I received the following warnings:





When running pytest locally I received another warning:



It seems as though the 3rd party libraries that were used in my requirements.txt file had vulnerabilities or deprecated functions that I had no control over. I believe this is something that is faced frequently in industry as 3rd party libraries can be chained to many levels that the developer is not aware of. There are software composition analysis tools that can help identify the vulnerable libraries but it seems the only mitigation to this is to identify the dependencies and use alternate libraries which is not easy when found towards the end of the development cycle, making CI/CD integration even more important.