**Michael Tse**

**11/8/19**

**Application Security – Assignment 3 – Fall 2019**

**Github Repository -** <https://github.com/mt1836/Assignment3>

**INTRODUCTION**

In this assignment we were tasked to integrate a database to our spell checker web application from Assignment 2 that would enable persistent storage of user accounts, login/logout history and spell checking results from all submissions by each user. An administrator account is created during startup of the web application and its credentials are stored in the database. The administrator is able to query all user submissions/results as well as login/logout history. Users are only allowed access to their submission/result history and are not allowed to view any login/logout history. At the conclusion of the database integration, we attempt to perform a SQLi on our web app and mitigate any findings.

**DATABASE CREATION**

The database chosen for this Assignment was a SQLite DB using the SQLAlchemy extension for Flask. SQLAlchemy is an Object Relational Mapper (ORM) that translates Python classes to database tables and allows users to use function calls that are converted to SQL statements providing ease of use, and a secure and consistent implementation. Three tables were created using the classes defined below:

* **User-** This table/class stores all user account information which includes an auto generated id based on registration sequence, a username, phone number, password taken as form input during the registration process, a salt for salting passwords and two additional columns as back references for the remaining two tables in the database (post and login history)
* **Post**- This table/class stores all spell checking related data including an auto generated id, the submitted text to run against the spell checker, the results of the misspelled words, the date the text was submitted for spell checking, the number of spell check submissions by a user, and finally a user id foreign key that references the primary key id generated in the User table.
* **Login\_history –** This table/class stores login/logout activity to record session detail. It includes an auto generated id, a login and logout timestamp for a users session and finally a user id foreign key that references the primary key id generated in the User table.

**TEMPLATES**

To support the additional functionality, three html pages were created:

* **History.html** – Web page to display a users spell check history
* **Query\_details**.**html** – Web page to display details of a specific spell check submission/results
* **Login\_history**.**html** – Web page for the administrator to view the login/logout history for any user. Logging is critical in providing accountability/traceability if there is an incident that needs to be investigated.

**TESTING**

We needed to ensure that the additional functionality was implemented correctly and tested the following conditions:

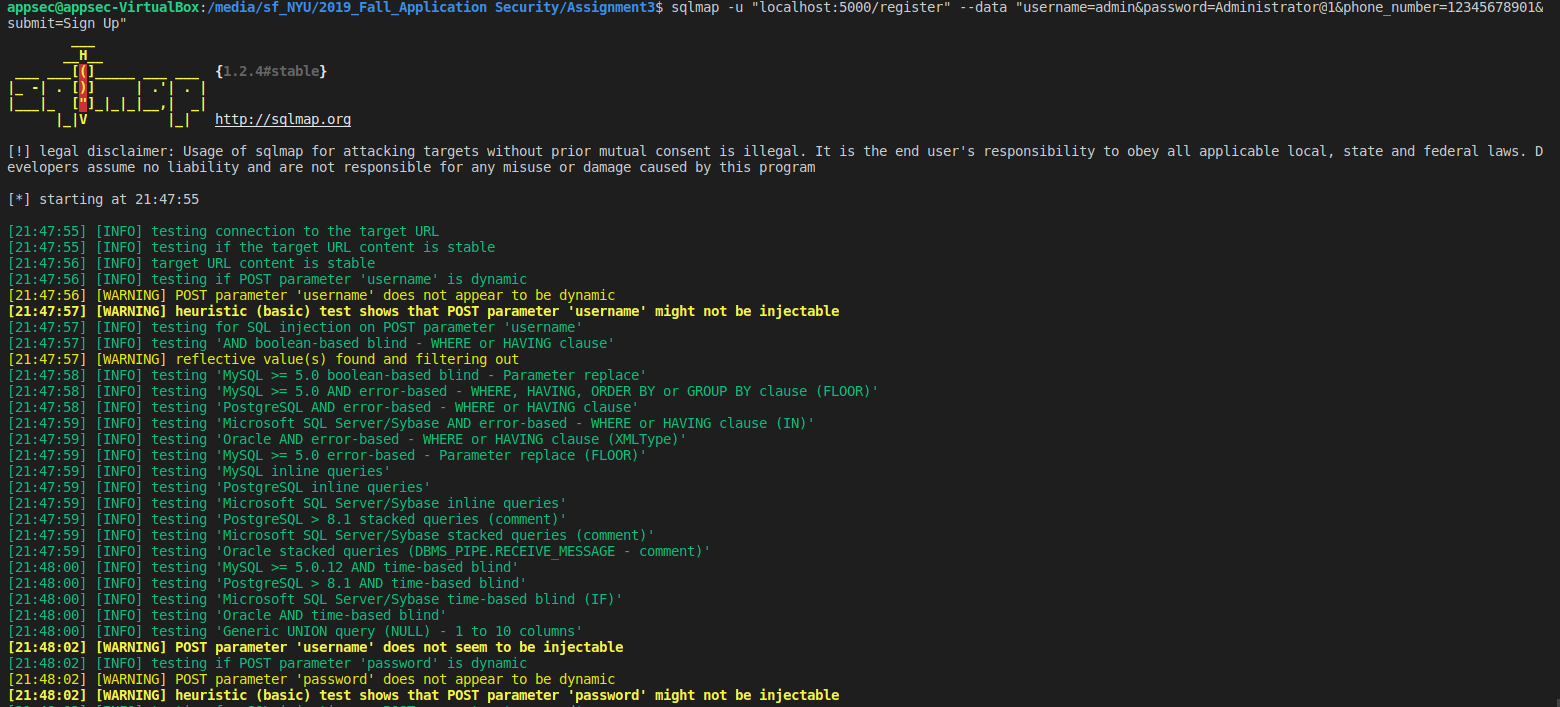
* Non-admin users are only allowed to view their own submission history - To address this we modified our login\_history.html to only display the input (user search) form to an administrator and added an if statement in our app.py file which only allows an admin to submit form data. For all other users the application automatically queries for the current username and queries the database for the necessary history information and returns it to history.html.
* Non-admin user is not able to view login history – This was performed on two fronts. First we eliminated the long history link from appearing when a non-admin user is logged in, however this does not restrict the user to manually manipulating the URL to get to the login history page. To address this we added an if statement in our app.py file which only allows an admin to submit form data and modified our login\_history.html to only display the input (user search) form to an administrator.
* Non-admin users should not be allowed to access query history by directly modifying the URL – It is possible for users to directly access query history by modifying the URL to go to specific query details (localhost:5000/history/query#). To address this we added the following code that checked who the owner was of the query#. If the owner is the current user then access was allowed. If the owner was not the current user a second check was performed to see if the current user was an admin. If they were not an admin they were not allowed access. If they were an admin they were allowed direct access via URL modification.

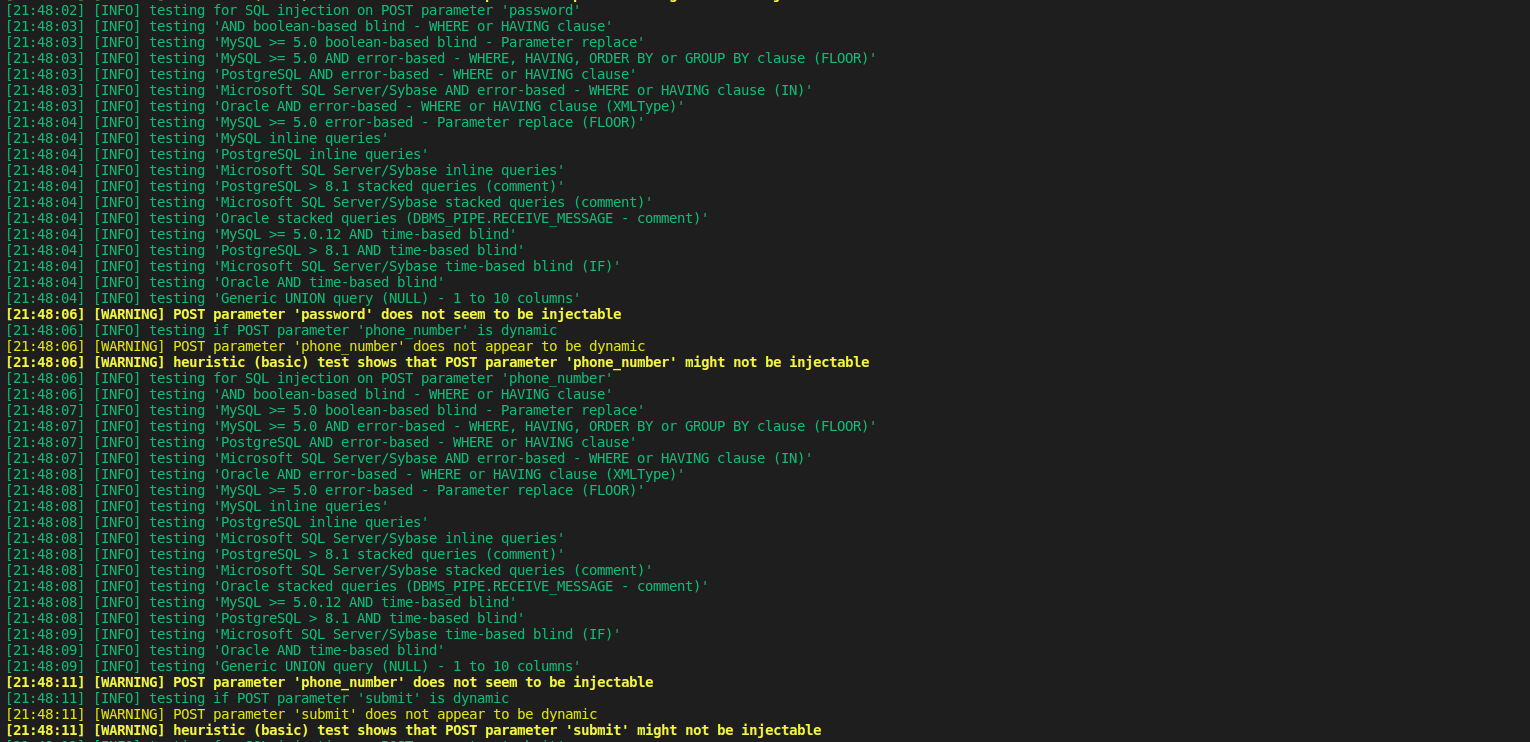
Since most of the testing for the web app was performed in the previous two assignments, the security test performed in this assignment was focused specifically on SQL injections. We used SQL Map as a tool to perform these tests. SQL Map is a piece of software that detects and exploits database vulnerabilities and automates the process of detecting and exploiting SQL injection flaws. It supports many databases and techniques and provides an easy and comprehensive way for developers to test their implementation.

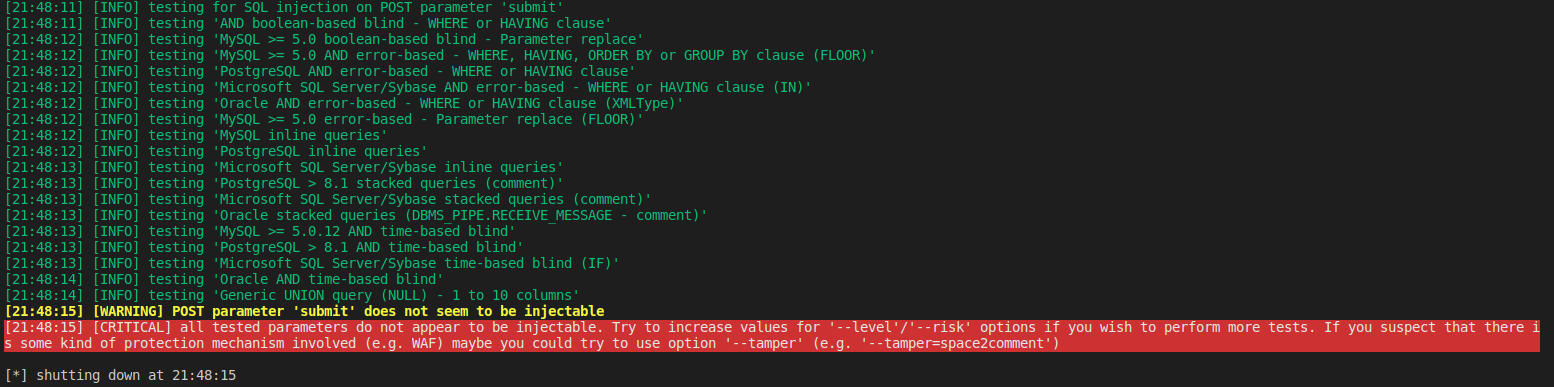
To test using SQL Map we first disabled CSRF from our application to reduce complexity as the SQL Map tool would need to have the CSRF token in order to run through its sequence of tests. We then entered the command below to initiate the test against the POST form input fields username, password and 2fa for our registration page:

The resulting output is shown below:

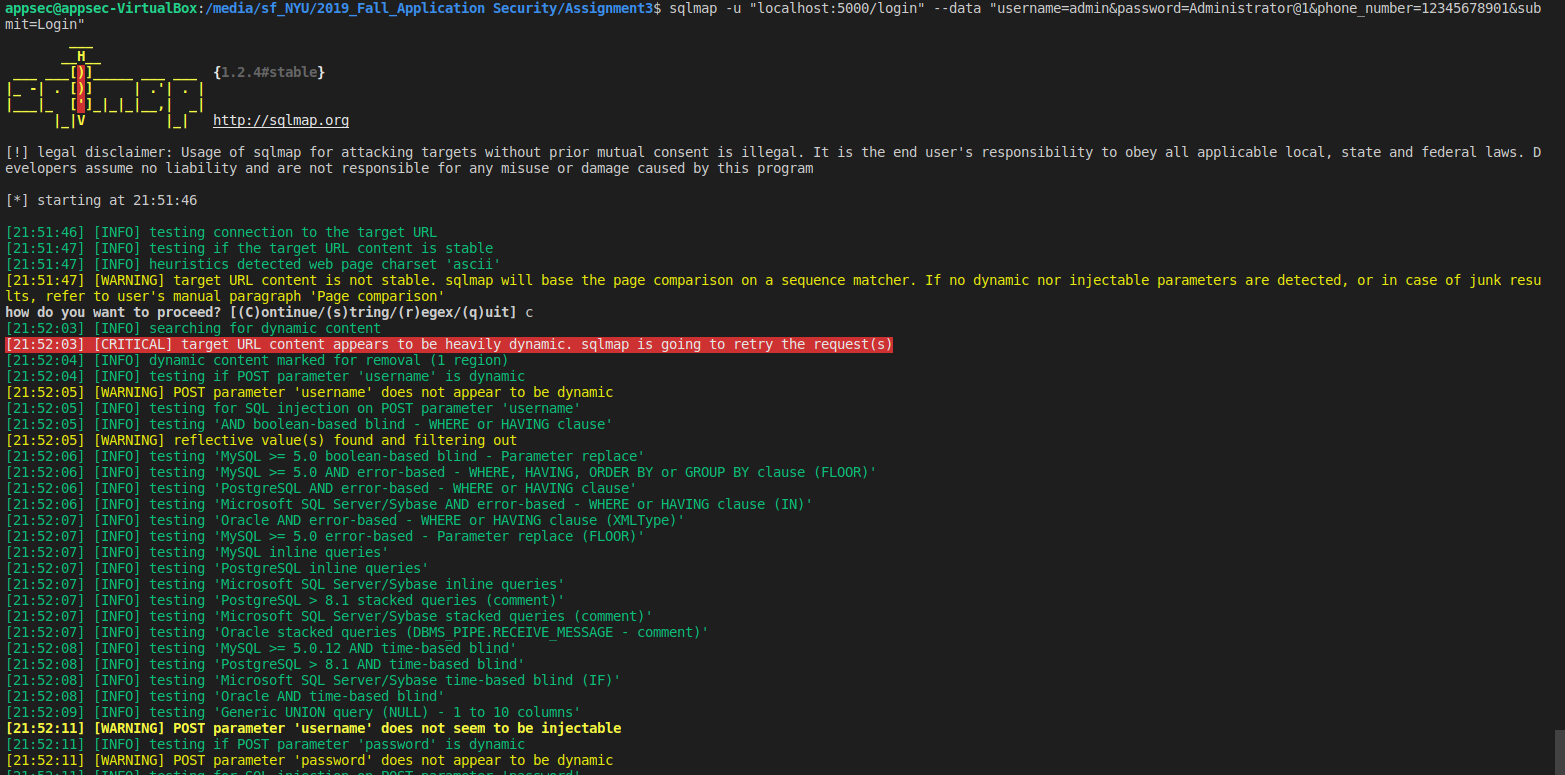


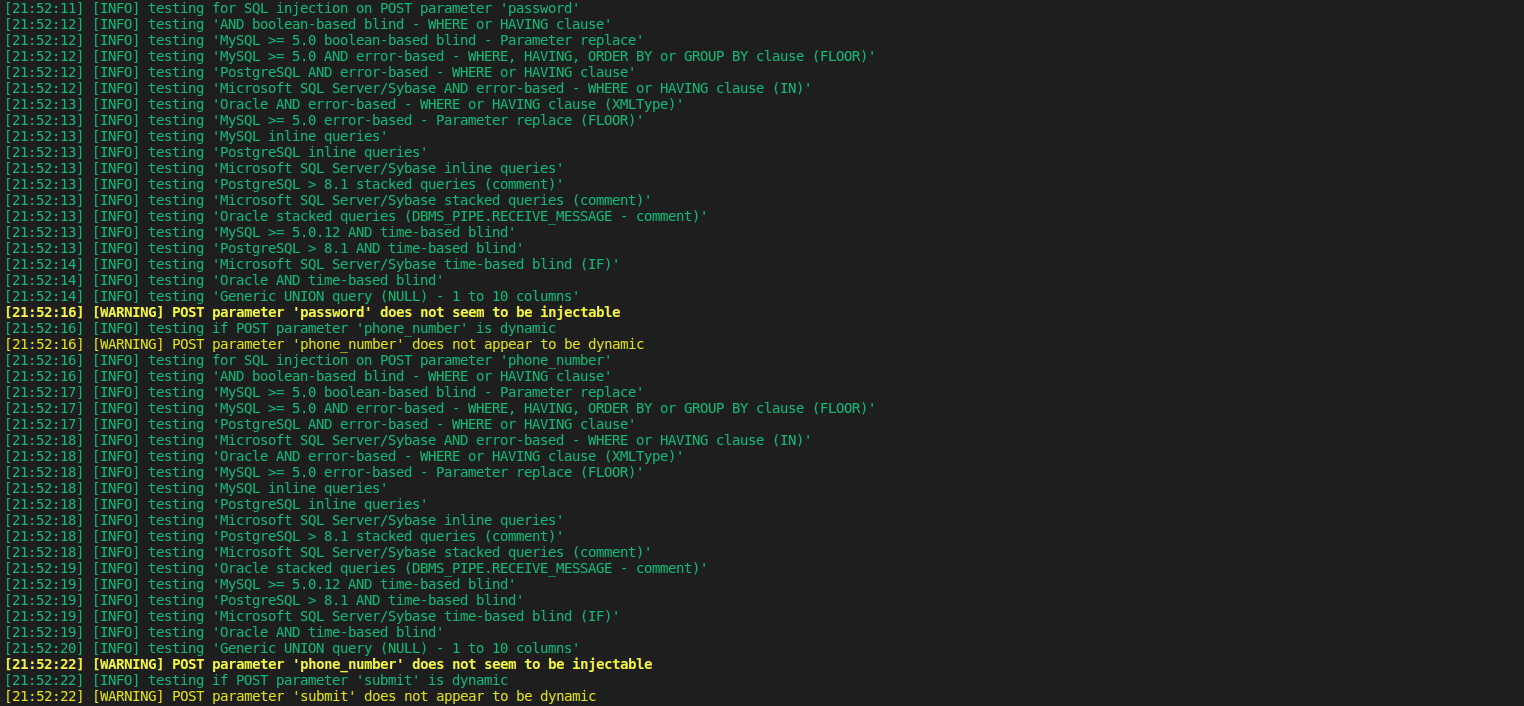


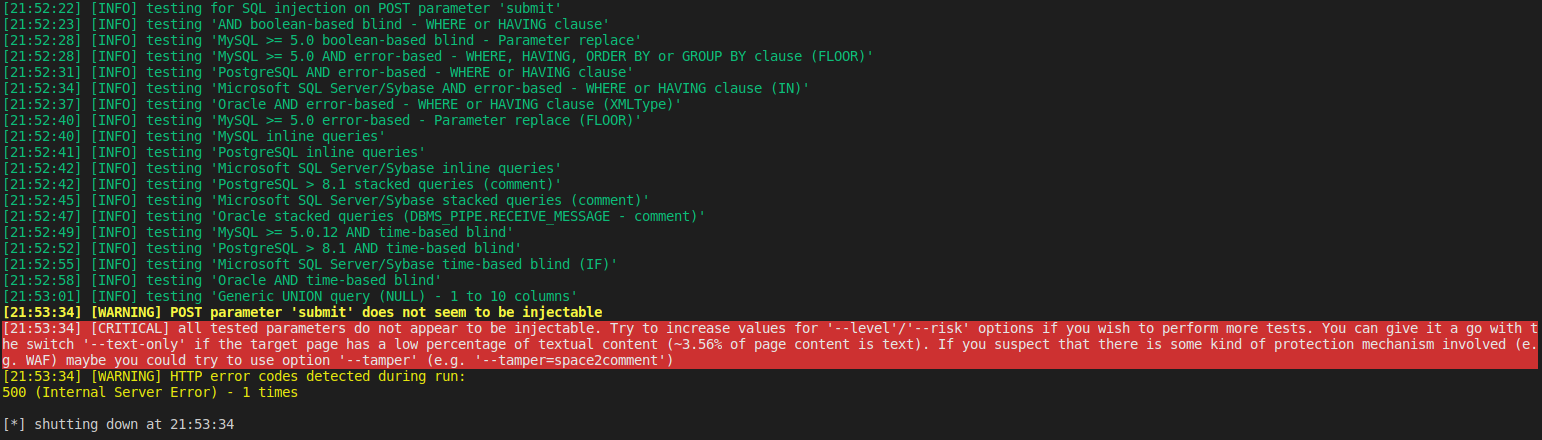




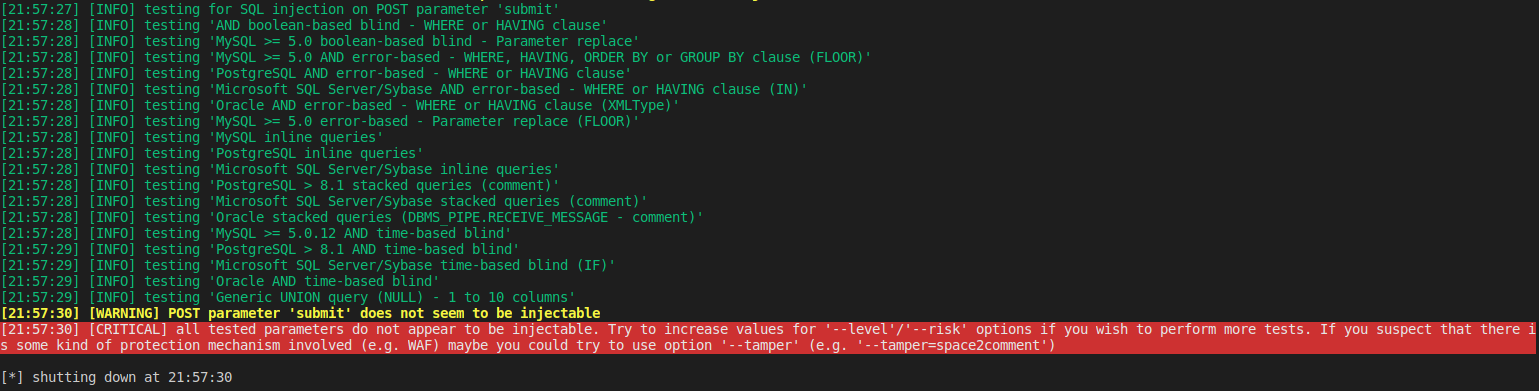
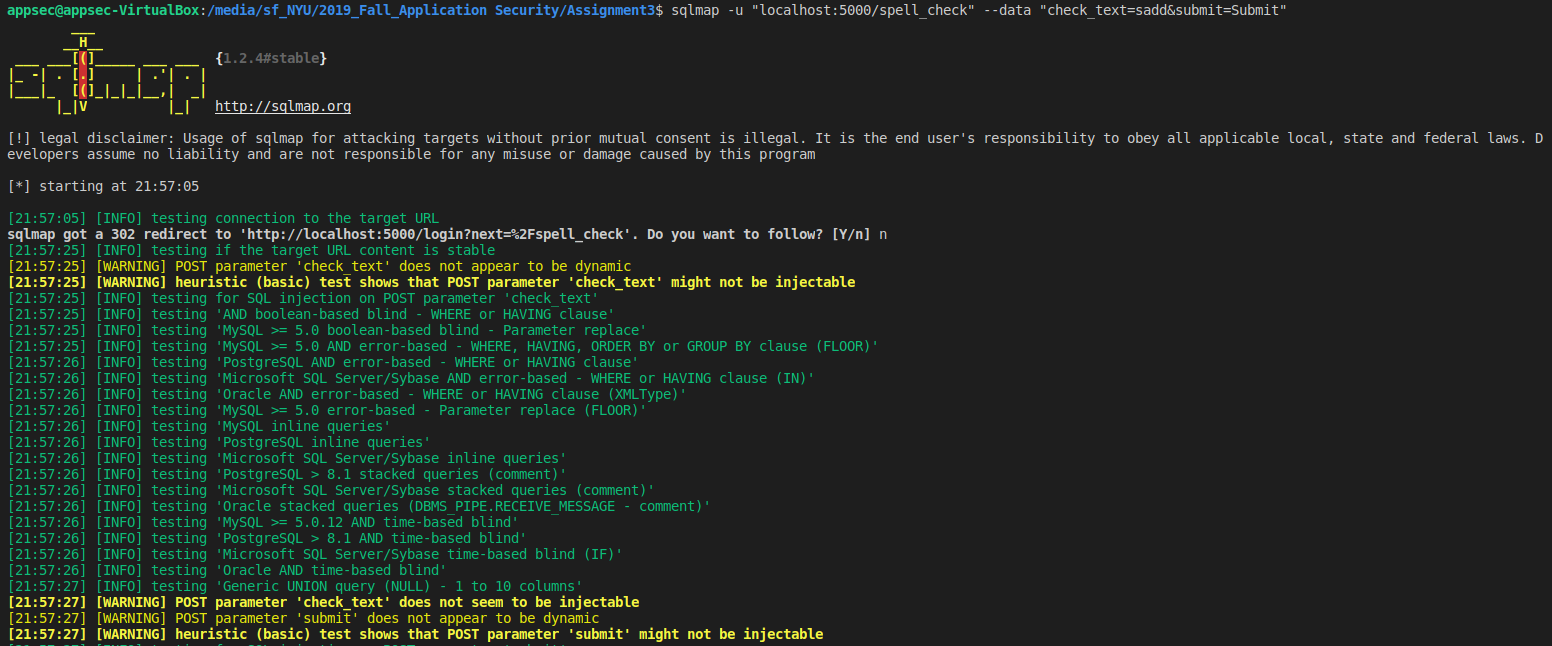
The same was done for the login page:



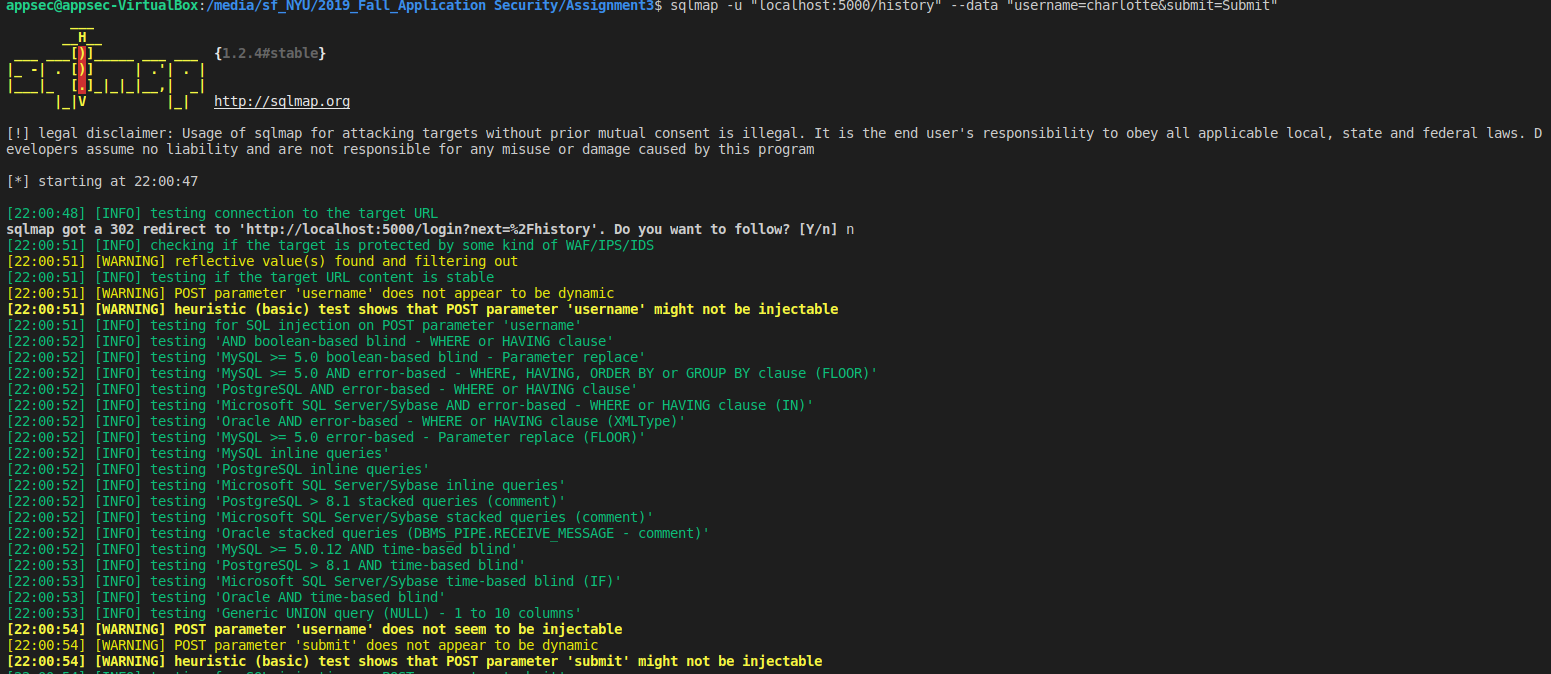


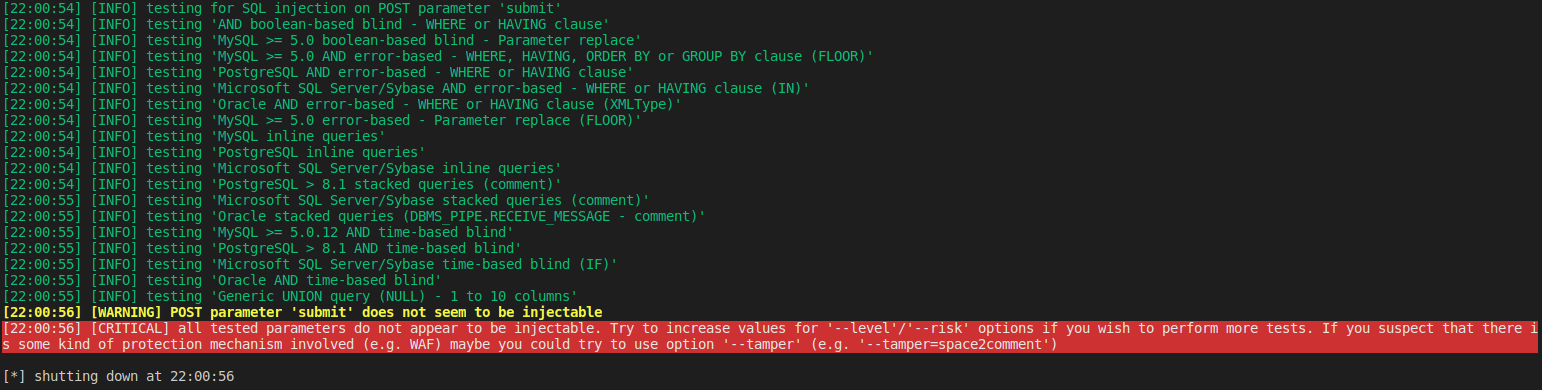


For the spell check page, we were required to log in first before running the SQL Map command. It resulted in the following:

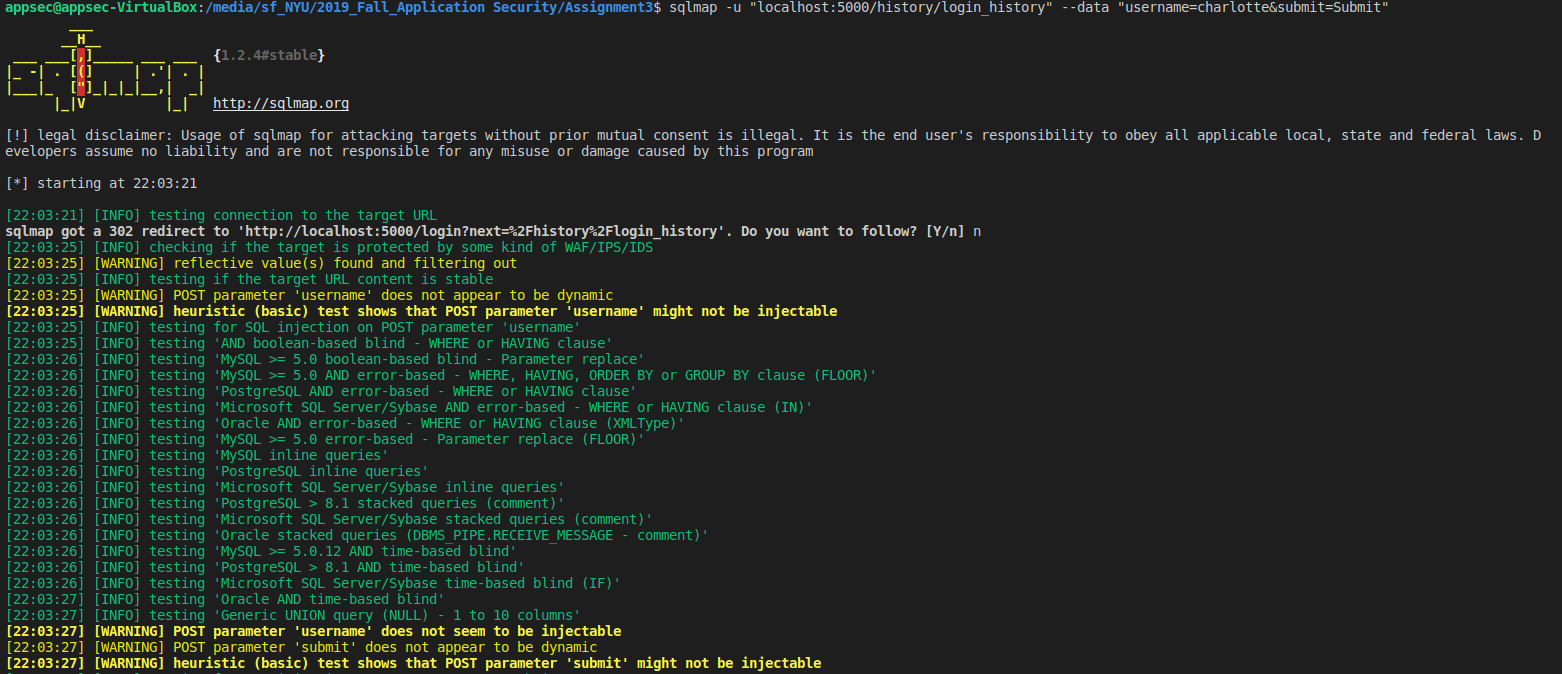
 

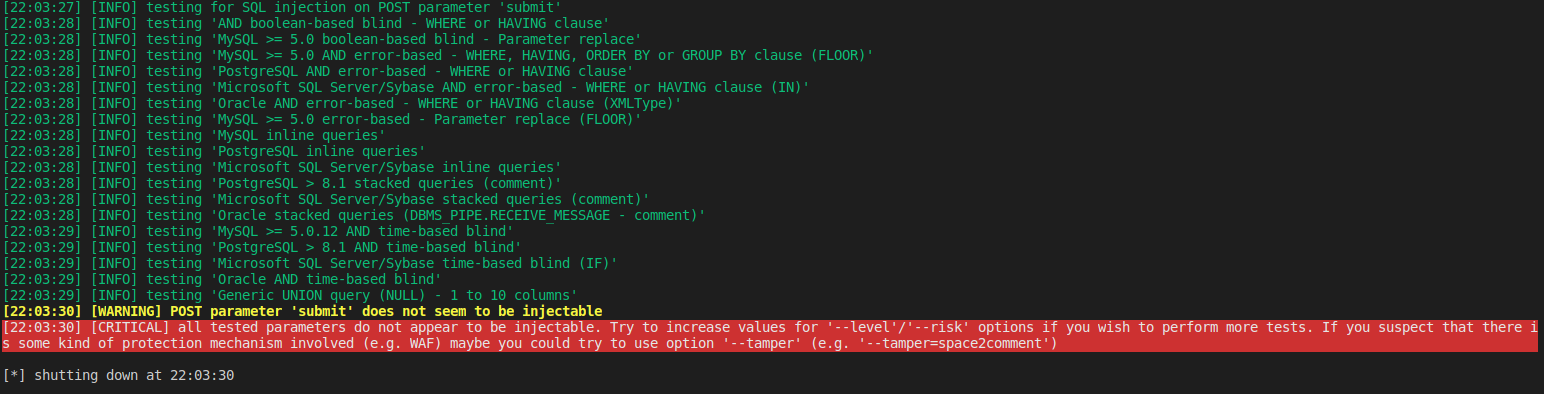
Same was performed for the history page. Results shown below:





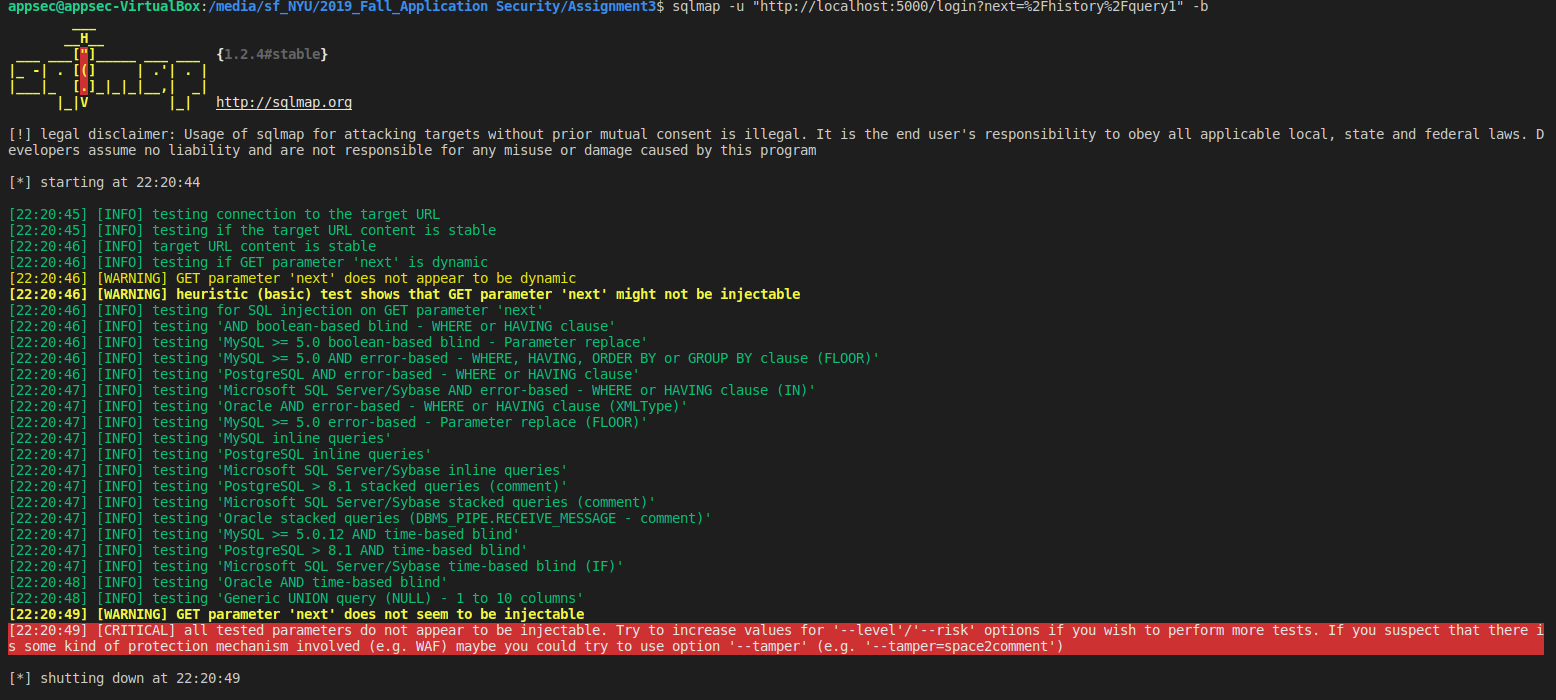
The login\_history page is the final page accepting form input to be POST tested. Results shown below:





The output of all POST tests showed that our web application was not likely vulnerable to SQLi. The final test was to check if our GET requests for the query# directly via the URL was vulnerable. We did this using the command and results below:





All of our GET and POST tests against user data entry points from our web application turned up to be unlikely for SQLi vulnerability. This is due to the SQLAlchemy ORM we used which not only provides ease of use but provides consistent implementation in a secure manner. The developers of the ORM are focused in secure design such that translated code into SQL statements are not subject to common vulnerabilities. For developers who choose not to use an ORM, they would need to understand in detail how SQLi work and the different types in order to develop applications in a secure manner. Even with this knowledge they cannot guarantee that they will not overlook something and code in a consistent manner. For these reasons ORMs provide developers an easy way to secure their applications from a majority of vulnerabilities and should be used where possible.