# Assignment 3: Web Security

This project is due on **October 22, 2021** at **5 pm**. You may work with a partner on this assignment and submit one project per team. You may NOT work with the same partner you have worked with on a previous assignment. Submit your solutions electronically.

---

## Background

A startup named **BUNGLE!** is about to launch its first product: a web search engine that accepts logins and tracks users' search histories. However, their investors are nervous about security problems. Unlike the Bunglers who developed the site, you took COSC 311, so the investors have hired you to perform a security evaluation before the site goes live to the public.

The **BUNGLE!** site is written in Python using the Bottle web framework. Although Bottle has built-in mechanisms that help guard against some common vulnerabilities, the Bunglers have circumvented or ignored these mechanisms in several places.

Your first task will be to attack the **BUNGLE!** website by exploiting three common classes of vulnerabilities: cross-site scripting (XSS), cross-site request forgery (CSRF), and SQL injection. You will need to exploit these vulnerabilities with various flawed defenses in place. Understanding how these attacks work will help you better defend your own web applications.

Your second task will be to modify the source code of the **BUNGLE!** website to defend against password breaches and some of the attacks you have demonstrated. Protect **BUNGLE!**... protect the world!

## Objectives

- Learn to spot common vulnerabilities in websites and to avoid them in your own projects.
- Understand the risks these problems pose and the weaknesses of naïve defenses.
- Gain experience with web architecture and with HTML, JavaScript, and SQL programming.

## Ethics Notice

This assignment asks you to develop and test attacks against a target website provided for this purpose. Attempting the same kinds of attacks against other websites without authorization is prohibited by law and university policies and may result in *disciplinary action, expulsion, fines, and/or jail time*. You must not attack any website without authorization!

# Provided Files

- `Readme.pdf`: This file.
- `Bungle.zip`: Zipped directory containing the source code of the **Bungle!** website. This includes the following files:
  - `bungle.py`, `database.py`, and `defenses.py`: Python files defining the website backend. These files implement the Bottle webserver, handle interactions with the database via SQL queries, and perform (insufficient) defenses against XSS and CSRF attacks.
  - `base.tpl`, `index.tpl`, and `search.tpl`: These template files define the website frontend. They are HTML files with template code that the backend fills in before they are served to users' browsers.
  - `auth.secret`: Text file with secret key used to set and access a cookie to preserve user logins across sessions.
  - `favicon.ico`: Image of the **Bungle!** logo for snazzy URL bars.
- `Solutions.txt`: File for you to complete with your solutions to Parts 1–4 and written responses to open-ended questions.

# Environment Setup

Before attempting attacks, you should first get the **Bungle!** website to run on your local machine. Take the following steps to run **Bungle!**:

- Run `pip3 install bottle` from your terminal to install the Bottle library.
- Run `pip3 install argon2-cffi` from your terminal to install the Argon2 library.
- Unzip the provided `Bungle.zip` into a directory of your choice.
- `cd` into the newly unzipped Bungle directory and run `python3 bungle.py` to launch a webserver hosting the **Bungle!** website for local access. The server will run until you press `ctrl-C` or it has an unrecoverable error.
- Open your a browser (Firefox strongly recommended) and type `http://0.0.0.0:8080` into the address bar. You should see the **Bungle!** homepage. Note that the Bunglers didn't even bother to implement TLS functionality, so you should see a crossed-out lock or other warning that your connection to **Bungle!** is insecure. This will obviously need to be fixed before launch day, but since you've only been hired to find XSS, CSRF, and SQL injection vulnerabilities, you don't need to worry about the lack of encryption.
- Play around with the **Bungle!** site as a non-malicious user and examine the source code to understand how it works. Note that the Bunglers haven't yet added search results to the database, so all searches will return "No results found."
- If you want to reset the database (remove all users and search histories) to start fresh, simply close the webserver and delete the `bungle.db` file.

# Bungle! Technical Guide

The **BUNGLE!** website is implemented using the Python `Bottle` library. The site also uses the built-in Python `SQLite3` library to manage a SQL database with two tables: 1) a `users` table with usernames and passwords and 2) a `history` table that maps usernames to prior searches. These tables are stored in the `bungle.db` file. **Your attacks may not attempt to directly open and modify `bungle.db`.** Assume that this file will be replaced with a remote database when **BUNGLE!** launches and all interactions with the database must go through SQL queries.

The **BUNGLE!** site replies to six main URLs: `/`, `/search`, `/create`, `/login`, `/logout`, and `/clear`. The function of these URLs is explained below:

**Main page (`/`)** The main page accepts `GET` requests and displays a search form. When submitted, the search form issues a `GET` request to `/search`, sending the search string as the parameter "`q`". If no user is logged in, the main page also displays a form that gives the user the option of logging in or creating an account. This form issues `POST` requests to `/login` and `/create`.

**Search results (`/search`)** The search results page accepts `GET` requests and prints the search string, supplied in the "`q`" query parameter, along with the search results. If the user is logged in, the page also displays the user's recent search history in a sidebar.

**Create account handler (`/create`)** The create account handler accepts `POST` requests and receives plaintext "`username`" and "`password`" query parameters. It inserts the username and password into the database of users, unless a user with that username already exists. It then logs the user in and redirects the browser to the main page. User passwords are neither sent nor stored securely, but your attacks should not exploit this lack of encryption.

**Login handler (`/login`)** The login handler accepts `POST` requests and takes plaintext "`username`" and "`password`" query parameters. It checks the user database to see if a user with those credentials exists. If so, it sets a login cookie and redirects the browser to the main page. The cookie tracks which user is logged in. **Your attacks may not modify the login cookie.**

**Logout handler (`/logout`)** The logout handler accepts `POST` requests. It deletes the login cookie, if set, and redirects the browser to the main page.

**Clear history (`/clear`)** The clear handler accepts `POST` requests and clears the search history of the currently logged-in user.

The Bunglers have been experimenting with some naïve defenses, which you will need to demonstrate provide insufficient protection. The site includes drop-down menus that let you change the defenses that are in use. You can also specify which defense level to use in URL query strings. For example, the following URL would access the **BUNGLE!** homepage (`/`) with XSS defense level 2 and CSRF defense level 1:

- `http://0.0.0.0:8080/?xssdefense=2&csrfdefense=1`

**Your attacks may not attempt to subvert the mechanism for changing the defense level.** Consider the defense level as a parameter that would be fixed at launch and unavailable to an attacker.

# Resources

The following resources will be useful for this assignment:

**Web Developer Tools.** Your browser's web developer tools will be *extremely* helpful for this project, particularly the JavaScript console and debugger, DOM inspector, and network monitor. To open these tools in Firefox, see the "Web Developer" options under "Tools" in the menu bar. Documentation here: `https://developer.mozilla.org/en-US/docs/Tools`. All major browsers have developer tools, so using Firefox is not required, but if you are having problems accessing or testing **BUNGLE!** or getting one of your attacks to work, try a different browser.

**Library Documentation.** You may wish to refer to the Bottle documentation to understand how the **BUNGLE!** backend handles HTTP requests and responses. You may also wish to refer to the Sqlite3 documentation to understand how the Python backend interacts with the database:

- **Bottle:** `https://bottlepy.org/docs/dev`
- **Sqlite3:** `https://docs.python.org/3/library/sqlite3.html`

**Web Programming Resources.** Your solutions will involve manipulating SQL statements and writing web code using HTML, JavaScript, and the jQuery library. You should search the web for answers to how-to questions. There are many good online resources for learning these tools. Here are a few recommendations:

- **Mozilla:** `https://developer.mozilla.org/en-US/docs/Learn`
- **W3Schools:** `https://www.w3schools.com/`
- **jQuery Learning Center:** `https://learn.jquery.com/`

**Attack References.** The OWASP website has useful references about XSS, CSRF, and SQL injection attacks (`https://owasp.org/www-community/attacks/`) and cheatsheets for defending websites (`https://cheatsheetseries.owasp.org/`). You should also search the web for suggestions and cite your primary sources in `Solutions.txt`.

**Attack Limits Reminder.** Your attacks may NOT

- Attempt to directly open and modify the `bungle.db` file
- Attempt to modify the login cookie
- Attempt to subvert the mechanism for changing the defense level
- Leverage the fact that the webserver is running locally to disrupt the server or modify the `.py` backend files.

# Part 1. Cross-site Scripting (35%)

Your first goal is to construct XSS attacks against the **BUNGLE!** search box, which does not properly filter or escape user-entered search terms. In an attempt to provide some security, the Bunglers have implemented four XSS defenses. For each defense, your goal is to construct a URL that, when loaded by a victim's browser, executes a malicious payload.

**Deliverables:** For each of the defenses below, fill in the corresponding section of the `Solutions.txt` file with the **simplest** search query URL you can construct that bypasses the defense and executes the payload described on the next page. You should try to use different payload encoding techniques against each defense. **Do NOT only find a URL that attacks the toughest defense and submit that solution for all defense levels.**

**Tips:**

- Start by looking at the `search.tpl` file to see how search queries get inserted into the DOM.
- Some special characters (including ";" and " ") need to be percent-encoded in URLs: `https://en.wikipedia.org/wiki/Percent-encoding`.
- DOM elements may not be able to be accessed by a script until the DOM has finished loading.
- You can test JavaScript/jQuery in the console of your browser's web developer tools.
- The more difficult defenses require thinking outside the box to defeat. Consider how can you get around the restrictions to still inject valid code.

## XSS Defenses

### 1.0 No defense

For 1.0 only, also submit a human-readable version of your payload code (as opposed to the form encoded into the URL) in the corresponding location in `Solutions.txt`

### 1.1 Remove "script"

```
filtered_search_query = re.sub(r"(?i)script", "", query)
```

### 1.2 Remove several tags

```
filtered_search_query = re.sub(r"(?i)script|<img|<body|<style|<meta|
                               <embed|<object", "", query)
```

### 1.3 Remove `" '` and `;`

```
filtered_search_query = re.sub(r"[;'\"]", "", query)
```

### 1.4 [Optional extra credit] Encode < and >

```
filtered_search_query = query.replace("<", "&lt;").replace(">", "&gt;")
```

This challenge is difficult and may require finding a bug in the website or a 0-day vulnerability.

## XSS Payload

The payload (the injected code that your XSS attacks execute) should steal the **username** and the **most recent non-XSS search query** that a logged-in victim has performed on the **BUNGLE!** site. When a victim visits the URLs you create in `Solutions.txt`, their username and most recent non-XSS search query should be sent to a "malicious server" for collection. More specifically, your XSS attack should cause the victim's browser to report the stolen information to the "attacker" by issuing a GET request for the following URL:

- `http://localhost:31337/stolen?user=USERNAME&last_search=LAST_SEARCH`

The response to this GET request is irrelevant. As long as the request is sent, you can assume that the attacker received the message.

The payload may

- Include inline JavaScript
- Use jQuery, as it is loaded automatically by **BUNGLE!** (see `base.tpl`)

Otherwise, your code must be self contained.

You can test your XSS attacks by loading each attack URL as if you were a logged-in victim with a search history and using the developer tools in your browser to view the above GET request.

**It is recommended that you test your XSS attacks with a simple payload (e.g., `alert(0);`) before moving on to the full payload.**

# Part 2. Cross-site Request Forgery (20%)

Your second goal is to construct CSRF attacks against the login form that cause the victim to log in to an account you control, thus allowing you to monitor the victim's search queries by viewing the search history for this account.

**Deliverables:** For each of the defenses below, fill in the corresponding section of the `Solutions.txt` file with HTML that accomplishes a CSRF attack against the specified **BUNGLE!** target URL. When opened by a victim as standalone `.html` files, your HTML should log the victim's browser into **BUNGLE!** under the account "`attacker`" and password "`URpwn3d`".

**Tips:** Start by creating an account with username `attacker` and password `URpwn3d` so you can test your attacks. Remember that there is more than one way to have a form submitted or a link clicked...

**2.0 No defense**
*Target:* `http://0.0.0.0:8080/login?csrfdefense=0&xssdefense=4`

**2.1 Token validation (with XSS)**
With the token validation defense turned on, the server sets a cookie named `csrf_token` to a random 16-byte value and includes this value as a hidden field in the login form. When the form is submitted, the server verifies that the client's cookie matches the value in the form. *You are allowed to exploit the XSS vulnerability from Part 1.0 for this attack.*

*Target:* `http://0.0.0.0:8080/login?csrfdefense=1&xssdefense=0`

**2.2 [Optional extra credit] Token validation (without XSS)**
Accomplish the same task as in 2.1 without using XSS. This challenge is difficult and probably requires finding a bug in the provided code or a 0-day vulnerability.

*Target:* `http://0.0.0.0:8080/login?csrfdefense=1&xssdefense=4`

The HTML you submit may

- Embed inline JavaScript
- Load jQuery as a remote script:
  `<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>`
  and use the jQuery library that is already loaded by **BUNGLE!**.
- Use the JS Cookies library that is already loaded by **BUNGLE!**. Documentation here:
  `https://github.com/js-cookie/js-cookie/blob/master/README.md`

Otherwise, your code must be self contained. Test your solutions by opening them as standalone `.html` files while the **BUNGLE!** Python server is running.

## Part 3. SQL Injection (10%)

Your third goal is to demonstrate a SQL injection attack that will log you in as an arbitrary user without knowing the password.

**Deliverables:** Complete the corresponding section of the `Solutions.txt` file with inputs to the username and password fields on the **BUNGLE!** login form that will successfully log you in as user "`victim`" without needing to know the user's real password.

**Tips:** Start by creating an account with username `victim` and an arbitrary password so you can test your attacks. Then examine the `validateUser()` function in `database.py`.

## Part 4. Better Defenses (35%)

You have now demonstrated that the **BUNGLE!** website is terribly insecure. Your final goal, as heroic security engineers, is to take the following steps to improve the **BUNGLE!** source code and limit future attacks:

1. Make a **copy** of the entire `bungle` directory and rename it `bungle_secure`
2. The **BUNGLE!** password storage is very insecure. Use the Argon2 library (`https://argon2-cffi.readthedocs.io/en/stable/`) to add password salting and hashing to `database.py` in `bungle_secure` to protect users' passwords in case of future breaches. Remember that when you change the database schema in `database.py`, you need to delete `bungle.db` so the database can be reinitialized when the webserver restarts.
3. Modify `database.py` in `bungle_secure` to defend against SQL injection attacks, such as the one you performed in Part 3. This should be a one-line modification to the code.
4. Answer the open-ended questions in `Solutions.txt`

## Deliverables

Upload the following files to Moodle:

- Completed `Solutions.txt` with all places marked "TODO" filled in with your solutions.
- Compressed directory `bungle_secure.zip` with your newly secure implementation of the **BUNGLE!** website from Part 4.

## Extra Credit "Bug Bounty"

If you find a bug anywhere in this assignment that is not related to an intentional **BUNGLE!** security vulnerability, please inform Prof. Apthorpe. The first student (or partners) to find any particular bug will be given a small amount of extra credit. This is an incentive to start the assignment early and will help make the course better for students in future years.