Project 2: Web Security Pitfalls

This project is split into two parts, with the first checkpoint due on **Feb 23, 2023** at **6:00 pm** and the second checkpoint due on **Mar 2, 2023** at **6:00 pm**. We strongly recommend that you get started early.

- This is a group project; you are allowed and recommended to work in **teams of two**. If so, both partners **MUST** populate their `WebSec/partners.txt` file to show BOTH partners' NetIDs.
- Solutions MUST be submitted via git.
- Solutions MUST be committed AND pushed to **default branch of your repository (main)** prior to the deadline.
- Solutions MUST follow the formatting guidelines and use the template files we provide. All solutions MUST be in a directory called WebSec.

Any of your submissions must be entirely your own work, and you are bound by the Student Code. You MAY consult with other students about the conceptualization of the project and the meaning of the questions, but you MUST NOT look at any part of someone else's solution or collaborate with anyone outside your team. You may consult published references, provided that you appropriately cite them in the `works_cited.txt` file we provide.

"If someone else can run arbitrary code on your computer, it's not YOUR computer any more."
– Rich Kulawiec

Introduction

In the first checkpoint of this project, you are provided with a code skeleton of a simple web application. You are asked to complete tasks which include writing a SQL query script to construct a database, completing prepared statements, writing input filters and implementing token validation mechanism. In the second checkpoint of this project, we provide an insecure version of this website, and your job is to attack it by exploiting three common classes of vulnerabilities: cross-site scripting (XSS), cross-site request forgery (CSRF), and SQL injection (SQLi). You are also asked to exploit these problems with various flawed defenses in place. Understanding how these attacks work will help you better defend your own web applications.

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 Python's Bottle Framework, HTML, JavaScript, and SQL programming.

Read This First

This project asks you to develop attacks and test them, with our permission, against a target website that we are providing 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 *fines, expulsion, and jail time*. **You must not attack any website without authorization!** Per the course ethics policy, you are required to respect the privacy and property rights of others at all times, *or else you will fail the course.* See the "Ethics, Law, and University Policies" section on the course website.

General Guidelines

You SHOULD develop this project targeting Firefox 71 (the Firefox version on the VM). Different browsers include different client-side defenses against XSS and CSRF that may interfere with your testing. For consistency, evaluation will be performed in Firefox 71 on the class VM.

For your convenience during manual testing, we have included drop-down menus at the top of each page that let you change the CSRF and XSS defenses that are in use. However, your solutions MUST override these selections by including the `csrfdefense=`*n* or `xssdefense=`*n* parameter in the target URL, as specified in each task below. Solutions that rely on changing the intended protection level for that problem are **INVALID**.

Resources

The Firefox Web Developer tools will be a tremendous help for this project, especially the JavaScript console and debugger, DOM inspector, and network monitor. See `https://developer.mozilla.org/en-US/docs/Tools`.

You **MUST NOT** use tools that are designed to automatically test for vulnerabilities.

Your solutions will involve manipulating SQL statements and writing web code using HTML, JavaScript, and the jQuery library. Feel free to search the web for answers to basic how-to questions. There are many fine online resources for learning these tools.

Here are a few that we recommend *(be aware that contents of websites listed here may be updated and incompatible with the project's version)*:

Bottle Framework Tutorial:
`http://bottlepy.org/docs/dev/tutorial.html`
SQL Tutorial:
`http://www.w3schools.com/sql/`
SQL Statement Syntax:
`https://dev.mysql.com/doc/refman/5.6/en/dynindex-statement.html`
pyMySQL Documentation:
`https://pymysql.readthedocs.io/en/latest/`
Introduction to HTML:
`https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Introduction`
HTTP Made Really Easy:
`http://www.jmarshall.com/easy/http/`
Using jQuery Core:
`http://learn.jquery.com/using-jquery-core/`
jQuery API Reference:
`http://api.jquery.com`

To learn more about SQL Injection, XSS, and CSRF attacks, and for tips on exploiting them, see:

`https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet`
`https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet`
`https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet`
`https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet`
`https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OTG-INPVAL-005)`

Target Website

A startup named **BUNGLE!** is about to launch its first product—a web search engine—but their investors are nervous about security problems. The Bunglers seem to have overlooked protections against several common attacks. Unlike the Bunglers who developed the site, you took CS 461/ECE 422, so the investors have hired you to perform a security evaluation before it goes live.

BUNGLE! is available for you to test at `http://bungle-cs461.csl.illinois.edu`. **Note:** You can only access **BUNGLE!** from the campus network or through the school VPN.

The site does not return any search results yet. But, the site accepts logins and tracks users' search histories. It stores usernames, passwords, and search history in a MySQL database.

Before being granted access to the source code, you reverse engineered the site and determined that it replies to five main URLs: `/`, `/search`, `/login`, `/logout`, and `/create`. The function of these URLs is explained below, but if you want an additional challenge, you can skip the rest of this section and do the reverse engineering yourself.

Main page (`/`) The main page accepts `GET` requests and displays a search form. When submitted, this 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. The 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.

Note: Since actual search results are not relevant to this project, you will not receive any results.

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; manipulating or forging it is **not** a part of this project.

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

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.

2.1 Checkpoint 1 (20 points)

Before you examine the real **Bungle!** written by Bunglers, you will implement some parts of **Bungle!**. This will help you understand its functionalities, security mechanisms, and potential vulnerabilities. You will use Python 3 (Bottle Framework and MySQLdb) and SQL for this checkpoint. The correct version of python3 libraries and MySQL are already installed on the VM.

2.1.1 VM Setup:

We use the same VM throughout the semester.

Host Bungle! Yourself

The code skeleton for **Bungle!** is available in your GitHub repository. You need to complete some functionalities in your locally hosted version of **Bungle!** as an introduction to this project. After cloning your repo on the VM, you can start a local server within the `<repo>/WebSec/bungle` directory with the following command and connect to the server at `http://127.0.0.1:8080/`.

```
python3 project2.py
```

Most parts of the **Bungle!** server are not functional yet, since you have not implemented them. In the following sections, you will implement each part of **Bungle!**.

2.1.2 SQL (5 points)

2.1.2.1 Database and User Creation

In this section, you will write a script consisting of SQL queries. Running this script will construct a database for **Bungle!**. Before you begin writing this script, you MUST create a database and a user for this database. Below are the requirements for **Bungle!**'s database and its user.
You need not submit the database and user creation queries.

- The name of the database is **project2**.
- The name of the user is your **netid** (If you are working as a team, the NetID MUST be that of the GitHub repository that this submission resides in). Create the user for the `localhost`. You MUST use the password of the user specified in a file named **dbrw.secret**. Do NOT change this password.
- User **netid** MUST only have **insert**, **update** and **select** privileges for tables in the **project2** database.

2.1.2.2 SQL Script Writing Exercise

After you have created the database and a user associated to it, you should write SQL queries in `2.1.2.txt` with requirements shown below.
Note: All columns for both tables MUST NOT allow **null**. **You will lose points if the columns don't explicitly state this.**

- Create a **table** named **users**. This table will store **BUNGLE!** users' account information. This table includes the following columns:

  id: Values are stored as type **int unsigned** (don't use INT(32) or INT(10)) and need to be **auto_incremented**. This column should also be the **primary key** for table **users**. This column stores a unique identification integer for each user.

  username: Values are stored as **varchar(32)**. This column should be an **unique index** since duplicate usernames MUST NOT be allowed.

  password: Values are stored as **varchar(64)**.

  passwordhash: Values are stored as **varchar(64)**.

- Create a **table** named **history** which stores each user's search history. This table includes the following columns:

  id: Values are stored as type **int unsigned** and needs to be **auto_incremented**. This column should also be the **primary key** for table **history**. This column stores a unique identification integer for each search history.

  user_id: Values in this column should be stored as an **int unsigned** and this column should be an **index** (not a FOREIGN KEY). This column represents the id number of the user who wrote the query.

  query: Values are stored as a **varchar(2048)**. This column stores the user's query input.

Remember that all of these columns should NOT allow **null**. After you write the script which creates the two tables, you can test your script *in the MySQL console* with the command shown below.

```
source 2.1.2.txt
```

`2.1.2.txt` MUST be formatted as a valid sequence of SQL statements. To minimize compatibility problems with the auto-grader, you SHOULD format each command of your solution as a single-line, valid SQL statement terminated with a semicolon (and newline).
You MUST only submit creation commands to make the tables described above. You MUST NOT include commands to use a database, drop a database, drop tables, create or delete users, or grant/flush user permissions.

Files

1. 2.1.2.txt: an empty .txt file in which you will write SQL queries

2.1.3 Prepared Statements (5 points)

In this section, you will utilize the pyMySQL API on **BUNGLE!** so that user inputs are processed and stored in the MySQL database via SQL queries. You will be editing `bungle/database.py` to connect to the database and issue commands.

For each missing function, write a prepared statement using `cur.execute()` that either puts data into the database or fetches data out. Use prepared statements so that when SQL queries are processed, **BUNGLE!** can distinguish what is data and what is code. This is a protection against SQL injection.

For more information regarding module `cursors` and function `execute()`, please refer to the PyMySQL API from the Resources page.

For the `getHistory` function, you MUST only return the last 15 queries that have been made in descending order of when the search was made (including duplicates).

Files

1. bungle/database.py: a python3 code skeleton for **BUNGLE!**'s SQL query processing

2.1.4 Input Sanitation (5 points)

Prepared statements protect **BUNGLE!** against SQL injections, but this mechanism does not filter against inputs which can be interpreted as HTML code. Thus, **BUNGLE!** is currently vulnerable to XSS attacks!
In this section, you will implement **BUNGLE!**'s input sanitation filters against XSS attacks. There are multiple possible filters you can implement to protect your application against XSS, but for this exercise we will encode < and >, which are the characters used for HTML tags. In the provided code skeleton, a "no defense" filter is implemented as class `XSSNone`. You will implement class `XSSEncodeAngles` which filters and encodes input < and > to `<` and `>` respectively.

Once you have completed implementing this filter you can run your local **BUNGLE!** and test it at `http://127.0.0.1:8080/?xssdefense=5&csrfdefense=0`.

Files

1. bungle/defenses.py: a python3 code skeleton for **BUNGLE!**'s defense mechanisms

2.1.5 Token Validation (5 points)

Finally, you will implement a token validation mechanism, to protect against CSRF. The **BUNGLE!** server sets a cookie named `csrf_token` to a random hexadecimal 16-byte value (so the length of the token is a 32-character hex string) and also 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. In defenses.py, you will implement this mechanism in a class named `CSRFToken`. The pseudo-code for this mechanism is shown below.

```
token ← request's cookie "csrf_token"
if token is None
    token ← a random 16 byte hexadecimal string
endif
token → response's cookie "csrf_token"
return token
```

- `init` receives two parameters, `request` and `response`.
- `request` represents a request from a user, and `response` represents a response from the server.
- Both objects contain a cookie named `csrf_token` as a private variable.
- You can retrieve this cookie from `request` using a getter function `get_cookie("csrf_token")`. If the user does not have that cookie, then this function will return `None`. Otherwise, it will return the 16-byte value which was previously generated.
- Likewise, you can set cookie for `response` to `value` using a setter function `set_cookie("csrf_token", value)`.

After you have implemented this mechanism, you can run your local **BUNGLE!** and test it at `http://127.0.0.1:8080/?xssdefense=0&csrfdefense=1`. To see if your solution blocks logins with incorrect tokens, try opening the browser developer console and changing the invisible csrf_token embedded in the login form. Then, try logging in to a valid account. Your defenses MUST detect this as a CSRF attack and block an otherwise valid login.

Files

1. bungle/defenses.py: a python3 code skeleton for **BUNGLE!**'s defense mechanisms

Checkpoint 1: Submission Checklist

The following blank files for checkpoint 1 has been created in your GitHub repository under the directory WebSec. Modify the solutions inside the WebSec directory and commit them to GitHub. **We will grade your repository's default (main) branch.**

Team Members

`partners.txt` : a text file containing netids of both members, one netid per line. Remember, both partners MUST have a copy of this file in the WebSec directory and pushed to the default branch.

Cite Your Sources

`works_cited.txt` : No particular style is required. Include the URL.

example content of partners.txt

```
netid1
netid2
```

Solution Format

example content of 2.1.2.txt

```
CREATE TABLE users ... ;
CREATE TABLE history ... ;
```

List of solution files that must be submitted for checkpoint 1

- `partners.txt`
- `2.1.2.txt`
- `bungle/database.py`
- `bungle/defenses.py`
- `dbrw.secret`

2.2 Checkpoint 2 (100 points)

In this checkpoint, you will identify and exploit vulnerabilities against the real **BUNGLE!** site.

2.2.1 SQL Injection (30 points)

2.2.1.1 No defenses (5pts)

Log in as "victim" using SQL injection. Note that you will have to put "victim" in the username field because the server uses that to determine what user you have logged in as.

Target: `http://bungle-cs461.csl.illinois.edu/sqlinject0/`

2.2.1.2 Simple escaping (5pts)

Log in again as "victim" using SQL injection. This time, user input is sanatized by replacing every single quote with two single quotes (In MySQL, adjacent strings concatentate).

Target: `http://bungle-cs461.csl.illinois.edu/sqlinject1/`

2.2.1.3 Perils of Raw Bytes (10pts)

Log in as "victim" using SQL injection. This time, passwords are hashed using md5 and the raw 16 bytes of output are placed between single quotes in a SQL query (using PHP string interpolation).

```
SELECT * from usertablename WHERE username = '<username>' AND
    password='<MD5(password)>'
```

Hint: MySQL strings permit unprintable raw bytes, but what happens when the raw output of md5 includes a single quote? You MUST include the code you used to find an injection as 2.2.1.3.tar.gz (not .tar.zx – use the tar utility).

Target: `http://bungle-cs461.csl.illinois.edu/sqlinject2/`

2.2.1.4 Spying With SQL Injection (10pts)

Target: `http://bungle-cs461.csl.illinois.edu/sqlinject3/` Steal information about this database using SQL injection. Learn about and use available MySQL functions and built-in tables.

1. What is the name of the database in use?
2. What is the version of MySQL in use?
3. List the tables in the current database from #1 as a comma-separated list
4. Follow the hint in the answer to #3 to find your secret string. Do NOT submit your partner's!

To prove you solved this problem yourself, include a newline-separated list of the URLs you used to learn the answers to these questions. Follow the sample format at the end of this document.

2.2.2 Cross-site Request Forgery (CSRF) (20 points)

Create two invisible pieces of HTML that perform a CSRF attack on Bungle and log in a victim as attacker. The user "attacker" has a password of "l33th4x" (the first character is a lowercase "L"). When a user who is not logged in on Bungle opens 2.2.2.1.html or 2.2.2.2.html and then visits Bungle, they should already be signed in as "attacker". Your attack MUST rely on invisible iframes, and thus the URL bar MUST not change. Your attack MUST work every time, and not make infinite requests.

2.2.2.1 No Defenses (10pts)

Use CSRF defense level 0 and XSS defense level 5.

Use these query parameters in your request URLs to ensure you attack the correct protection level: `?csrfdefense=0&xssdefense=5`

Start by using the developer tools to see how Bungle's login process works. Use what you learn to replicate a sign-in request. A response to a cross-origin response will not set any cookies, so you will need to target an iframe which will be able to read the response and set the cookies.

2.2.2.2 Token validation (10pts)

Use CSRF defense level 1 and XSS defense level 0.

Use these query parameters in your request URLs to ensure you attack the correct protection level: `?csrfdefense=1&xssdefense=0`

A CSRF token defense just like you implemented in checkpoint 1 is in place, preventing cross-origin requests. However, you can cheat and get around Same-Origin Policy restrictions by using XSS. If you are unfamiliar with Javascript, we suggest doing 2.2.3.* first, because experience there will make this part easier. Watch out for race conditions–does your attack rely on asynchronous web requests to have a temporal happens-before relationship?

2.2.3 Cross-site Scripting (XSS) (50 points)

Build malicious links in 2.2.3.html to take over Bungle, spy on users, and hide any evidence of an attack. Get your attack past various XSS filters.

2.2.3.1 XSS Practice (5pts)

Practice XSS by replacing the INSERT_ATTACK_HERE in the given URL with a malicious script injection that changes the "Click me" link to point to `ece.illinois.edu`. The link's text MUST NOT change (case sensitive).

2.2.3.2 XSS Defense Level 0: No Defenses

This level has no input sanitization defenses (just like the warmup). We've given you a sample attack in 2.2.3.html to get you started. You MUST extend the attack to have the following 3 properties:

1. Persistence: Use the provided proxy() function in 2.2.3.html to create the illusion of navigation no matter what button the user clicks. Use event listeners to take over all buttons, links, and forms (do not forget past search results and the Bungle logo link). Also allow use of the forward and backward buttons. To prove you are still in control at all times, the tab title MUST be set to "Not Bungle!" You can assume we won't test incorrect user/passwords or other errors. We also won't test if the user refreshes the page.

2. Spying: Send a message to a spying server when the user navigates to a new page, logs in, creates an account, clicks a forward/backward button, or logs out. Since creating the spying server is not part of the MP, you only need to send the requests to `http://localhost:31337/stolen?`. Include query parameters to specify:

- event=nav (for page navigation) or event=login (for logins or account creation) or event=logout (for logout).
- user=<username_here> if the user is logged in or is logging in
- url=<url_here> for nav events, tell us where the user navigated (absolute or relative links accepted)
- pass=<password_here> steal the password on login/create and report it here

For example, a user logging out would trigger two spying HTTP requests:

```
http://127.0.0.1:31337/stolen?event=logout&user=bob0012
http://127.0.0.1:31337/stolen?event=nav&url=/
```

3. Stealth: You MUST hide evidence of your attack in the URL bar for stealth using HTML5's history API. Watch out for double slashes in your paths. Bungle MUST look and act just like normal

even though your code is running. Slight differences in font-size or brief flashes of unstyled HTML are not an issue for this MP. Hide any suspicious entries in the search history (it is okay if this results in fewer than 15 results).

2.2.3.3 XSS Defense Level 1: Remove "script"

This level's filter removes any instance of the string "script" from any search (case insensitive). Reuse your attack from the previous problem, but get it past this defense.

2.2.3.4 XSS Defense Level 2: Recursively removing "script"

This level's filter recursively removes any instance of "script" from any search (case insensitive). Reuse your attack from the previous problem, but get it past this defense.

2.2.3.5 XSS Defense Level 3: Recursively Removing several tags

This level's filter recursively removes anything that matches this case insensitive regular expression: Reuse your attack from the previous problem, but get it past this defense.

```
(?i)script|<img|<image|<body|<style|<meta|<embed|<object
```

2.2.3.6 XSS Defense Level 4: Remove some punctuation

This level's filter removes all semicolons, single quotes, and double quotes from any search. Reuse your attack from the previous problem, but get it past this defense.

2.2.3.7 XSS Grading

The warmup is worth 5 points. The remaining 45 points are split across the 5 protection levels. You SHOULD re-use your same payload across all five levels. Every level your code runs on is an opportunity to earn another 9 points.

1. 4.5 points are available for persistence.
2. 2.5 points are available for spying.
3. 2.0 points are available for stealth.

Checkpoint 2: Submission Checklist

Inside your WebSec directory, you will have the auto-generated files listed below. Make sure that your answers for all tasks up to this point are submitted in the following files before **Mar 2, 2023** at **6:00 pm**:

Team Members

`partners.txt` : a text file containing netid of both members, one netid per line. Place the student's netid, whose directory contains your project submission, at the top of the file. Both partners need to push an identical partners.txt to each of their repositories, in the WebSec directory, on the default branch.

Cite Your Sources

`works_cited.txt` : a text file for you to cite the sources you used for this MP. No particular style is required. Include the URL.

example content of partners.txt

```
netid1
netid2
```

example content of 2.2.1.1.txt, 2.2.1.2.txt and 2.2.1.3.txt

```
username=victim&password=PASSWORD
```

example content of 2.2.1.4.txt

```
DB_NAME
DB_VERSION
TABLE_NAMES,COMMA,SEPARATED,LIST
A_SECRET_STRING

http://URL_FOR_PROBLEM_1
http://URL_FOR_PROBLEM_2
http://URL_FOR_PROBLEM_3
http://ANOTHER_URL_FOR_PROBLEM_3
http://URL_FOR_PROBLEM_4
http://ANOTHER_URL_FOR_PROBLEM_4
...
```

example content of 2.2.3.html

```
<!DOCTYPE html>
<html>
    ...
</html>
```

List of solution files that must be submitted for checkpoint 2

- `partners.txt`
- `works_cited.txt`
- 2.2.1.1.txt
- 2.2.1.2.txt
- 2.2.1.3.txt
- 2.2.1.3.tar.gz
- 2.2.1.4.txt
- 2.2.2.1.html
- 2.2.2.2.html
- 2.2.3.html

Common Errors

- Race Conditions in 2.2.2.*. Remember, if your attack requires multiple web requests to be answered in a certain order, you must ensure this ordering in your code.
- Failure to include 2.2.1.3.tar.gz, or using another archive type.
- Incomplete testing of 2.2.3.*. Make sure you can create users, log in, log out, and click on past search results. If you change your payload to attack harder defense levels, make sure the prior ones still work.