Project 2: Web Security

This project is due on **Wednesday, February 19, 2020** at **6 p.m.** and counts for 13% of your course grade. Late submissions will be penalized by 10% of the maximum attainable score, plus an additional 10% every 4 hours until received. Late work will not be accepted after the start of the next lab (of any section) following the day of the deadline. If you have a conflict due to travel, interviews, etc., please plan accordingly and turn in your project early.

The code and other answers you submit must be entirely your own work, and you are bound by the Honor Code. You may discuss the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone. You may consult published references, provided that you appropriately cite them (e.g., with program comments). Visit the course website for the full collaboration policy.

Solutions must be submitted electronically via your GitHub repo, except the writeup which must be submitted through Gradescope. We encourage you to verify the submission checklist below.

Introduction

In this project, we provide an insecure website, and your job is to attack it by exploiting three common classes of vulnerabilities: SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF). 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 naive defenses.
- Gain experience with web architecture and with 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 for more info.

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. Unlike the Bunglers who developed the site, you took EECS 388, so the investors have hired you to perform a security evaluation before it goes live.

BUNGLE! is available for you to test at https://project2.eecs388.org/.

The 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. If you wish, you can inspect the Python source code contained in bungle.tar.gz from the starter files, but this is not necessary to complete the project.

In addition to providing search results, the site accepts logins and tracks users' search histories. It stores usernames, passwords, and search history in a MySQL database.
Note: Passwords used on the **BUNGLE!** site may be used, in whole or in part, in subsequent assignments in the course. **Never use an important password to test an insecure site! This especially includes your personal passwords.**

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 is not relevant to this project, you might 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** 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.

Note: The password is neither sent nor stored securely; however, none of the attacks you implement should depend on this behavior. You should choose a password that other groups will not guess, **but again, never use an important password to test an insecure site!**

Guidelines

Defense Levels The Bunglers have been experimenting with some naïve defenses, so you also need to demonstrate that these provide insufficient protection. In Parts 2 and 3, the site includes drop-down menus at the top of each page that let you change the CSRF and XSS defenses that are in use. When you are testing your solution, ensure that **BUNGLE!** has the correct defense levels set. **You may not attempt to subvert the mechanism for changing the level of defense in your attacks.** Be sure to test your solutions with the appropriate defense levels! Additionally, be sure to include the defense levels as url parameters in any request to bungle!

In all parts, you should implement the simplest attack you can think of that defeats the given set of defenses. In other words, do not simply attack the highest level of defense and submit that attack as your solution for all defenses. You do not need to combine the vulnerabilities, unless explicitly stated.

Resources The Firefox and Chrome web developer tools will be very helpful for this project, particularly the JavaScript console and debugger, DOM inspector, and network monitor. See https://developer.mozilla.org/en-US/docs/Tools and https://developers.google.com/web/tools/chrome-devtools. Note that you may complete the project on any web browser of your choice, but we have only tested that it works on Chrome, Firefox, and Safari.

Although general purpose tools are permitted, you are **not** allowed to 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. You should 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:

SQL Tutorial	https://sqlzoo.net/wiki/SQL_Tutorial
SQL Statement Syntax	https://dev.mysql.com/doc/refman/5.6/en/sql-statements.html
Introduction to HTML	https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Introduction
JavaScript 101	https://hsablonniere.github.io/markleft/prezas/javascript-101.html#1.0
Using jQuery Core	https://learn.jquery.com/using-jquery-core/
jQuery API Reference	https://api.jquery.com
HTTP Made Really Easy	https://www.jmarshall.com/easy/http/

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

https://github.com/OWASP/CheatSheetSeries
https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

Part 1. SQL Injection

Your first goal is to demonstrate SQL injection attacks that log you in as an arbitrary user without knowing the password. In order to protect other students' accounts, we've made a series of separate login forms for you to attack that aren't part of the main **BUNGLE!** site. For each of the following defenses, provide inputs to the target login form that successfully log you in as the user "`victim`":

1.0 No defenses

Target: `https://project2.eecs388.org/sqlinject/0`
Submission: `sql_0.txt`

1.1 Simple escaping

The server escapes single quotes (`'`) in the inputs by replacing them with two single quotes.

Target: `https://project2.eecs388.org/sqlinject/1`
Submission: `sql_1.txt`

1.2 Escaping and Hashing

The server uses the following PHP code, which escapes the username and applies the MD5 hash function to the password.

```
if (isset($_POST['username']) and isset($_POST['password'])) {
    $username = mysql_real_escape_string($_POST['username']);
    $password = md5($_POST['password'], true);
    $sql_s = "SELECT * FROM users WHERE username='$username' and pw='$password'";
    $rs = mysql_query($sql_s);
    if (mysql_num_rows($rs) > 0) {
        echo "Login successful!";
    } else {
        echo "Incorrect username or password";
    }
}
```

This is more difficult than the previous two defenses. You will need to write a program to produce a working exploit. You can use any language you like, but we recommend Python 3.

Target: `https://project2.eecs388.org/sqlinject/2`
Submissions: `sql_2.txt` and a directory called `sql_2-src/`

1.3 The SQL [Extra credit]

This target uses a different database. Your job is to use SQL injection to retrieve:

(a) The name of the database
(b) The version of the SQL server
(c) All of the names of the tables in the database
(d) A secret string hidden in the database

Target: `https://project2.eecs388.org/sqlinject/3`
Submission: `sql_3.txt`

For this part, the text file you submit should start with a list of all the queries you made to learn the answers. Follow this with the values specified above, using this format:

```
QUERY
QUERY
QUERY
...

Name: DB name
Version: DB version string
Tables: comma separated names
Secret: secret string
```

What to submit For 1.0, 1.1, and 1.2, when you successfully log in as `victim`, the server will provide a URL-encoded version of your form inputs. Submit a text file with the specified filename containing only this line. For 1.2, also submit the source code for the program you wrote by placing it in the directory `sql_2-src`. For 1.3, submit a text file as specified.

Part 2. Cross-site Scripting (XSS)

Your next task is to demonstrate XSS attacks against the **Bungle!** search box, which does not properly filter search terms before echoing them to the results page. For each of the defenses below, your goal is to construct a URL that, when loaded in the victim's browser, correctly executes the specified payload. We recommend that you begin by testing with a simple payload (e.g., `alert(0);`), then move on to the full payload. Note that you should be able to implement the payload once, then use different means of encoding it to bypass the different defenses.

Note: jQuery is embedded on Bungle. Please do not reload it in your scripts for Part 2.

Payload

The payload (the code that the attack tries to execute) will be to steal the username and the most recent search the real user has performed on the **Bungle!** site. When a victim visits the URL you create, these stolen items should be sent to the attacker's server for collection.

For purposes of grading, your attack should report these events by loading the URL:
`http://localhost:31337/?stolen_user=`*username*`&last_search=`*last_search*

You can test receiving this data by running this command at the shell in your project directory:

```
$ python3 -m http.server -b 127.0.0.1 31337
```

and observing the HTTP GET request that your payload generates in the server log.

Defenses

There are five levels of defense. In each case, you should submit the simplest attack you can find that works against that defense; you should not simply attack the highest level and submit your solution for that level for every level. Try to use a different technique for each defense. The Python

code that implements each defense is shown below, along with the target URL and the filename you should submit.

2.0 No defenses

Target: `https://project2.eecs388.org/search?xssdefense=0`
Submission: `xss_0.txt`

For 2.0 only, also submit a human-readable version of your payload code (as opposed to the form encoded into the URL). Save it in a file named `xss_payload.html`.

2.1 Remove "script"

```
filtered = re.sub(r"(?i)script", "", input)
```

Target: `https://project2.eecs388.org/search?xssdefense=1`
Submission: `xss_1.txt`

2.2 Remove several tags

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

Target: `https://project2.eecs388.org/search?xssdefense=2`
Submission: `xss_2.txt`

2.3 Remove some punctuation

```
filtered = re.sub(r"[;'\"]", "", input)
```

Target: `https://project2.eecs388.org/search?xssdefense=3`
Submission: `xss_3.txt`

2.4 Encode < and > [Extra credit]

```
filtered = input.replace("<", "&lt;").replace(">", "&gt;")
```

Target: `https://project2.eecs388.org/search?xssdefense=4`
Submission: `xss_4.txt`

This challenge is hard. We think it requires finding a 0-day vuln or a bug in our code.

What to submit Your submission for each level of defense will be a text file with the specified filename that contains a single line consisting of a URL. When this URL is loaded in a victim's browser, it should execute the specified payload against the specified target. The payload encoded in your URLs may embed inline JavaScript.

Part 3. Cross-site Request Forgery (CSRF)

Your final goal is to demonstrate CSRF vulnerabilities against the login form, and **Bungle!** has provided two variations of their implementation for you to test. Your goal is to construct attacks that surreptitiously 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. For each of the defenses below, create an HTML file that, when opened by a victim, logs their browser into **Bungle!** under the account "`attacker`" and password "`l33th4x`". NOTE: the first character of the password is a letter, not a number.

Your solutions should not display evidence of an attack; the browser should just display a blank page. (If the victim later visits **Bungle!**, it will say "logged in as attacker", but that's fine for purposes of the project. After all, most users won't immediately notice.)

3.0 No defenses
Target: `https://project2.eecs388.org/login?csrfdefense=0&xssdefense=4`
Submission: `csrf_0.html`

3.1 Token validation
The server sets a cookie named `csrf_token` to a random 16-byte value 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. You are allowed to exploit the XSS vulnerability from Part 2 to accomplish your goal.

Target: `https://project2.eecs388.org/login?csrfdefense=1&xssdefense=0`
Submission: `csrf_1.html`

3.2 Token validation, without XSS [Extra credit]
Accomplish the same task as in 3.1 without using XSS.

Target: `https://project2.eecs388.org/login?csrfdefense=1&xssdefense=4`
Submission: `csrf_2.html`

This challenge is hard. We think it requires finding a 0-day vuln or a bug in our code.

What to submit For each part, submit an HTML file with the given name that accomplishes the specified attack against the specified target URL. The HTML files you submit may embed inline JavaScript.

Note: Since you're sharing the `attacker` account with other students, we've hard-coded it so the search history won't actually update. You can test with a different account you create to see the history change.

Part 4. Writeup: Better Defenses

For each of the three kinds of attacks (SQL injection, XSS, and CSRF), write a paragraph of advice for the **BUNGLE!** developers about what techniques they should use to defend themselves. A LaTeX template, `writeup.tex`, is provided in the starter files. Fill in the template and convert it to a PDF titled `writeup.pdf`. Please submit `writeup.pdf` to **Gradescope.**

What to submit A PDF file named `writeup.pdf` containing your security recommendations. Ensure that you have submitted it to Gradescope.

Submission Details

1. Create a repo using this GitHub Classroom link.
2. Submit your GitHub ID here so your submission can be matched to your uniqname.
3. Check that your repo contains the starter files. If you have any issues with GitHub Classroom, email `eecs388-staff@umich.edu` for assistance.
4. At each deadline, your repository will be cloned automatically for grading.

   If you need to submit late, you must report the hash of the commit you want graded using the late submission form. The late penalty will apply to the timestamp of the form submission, not the timestamp of the commit.

Where applicable, your solutions may contain embedded JavaScript. They must be self-contained, but may use the jQuery already loaded onto the page. You may not attempt to subvert the mechanism for changing the level of defense in your attacks.

Part 1: SQL Injection

For parts 1.0, 1.1, and 1.2, submit text files that contain the strings provided by the server when your exploits work. For 1.2, also submit a tarball containing the source code you wrote to produce your solution. For 1.3, submit a text file as specified in the problem statement. Make sure your text files are actual .txt files or we may not be able to grade your work.

`sql_0.txt`	1.0 No defenses
`sql_1.txt`	1.1 Simple escaping
`sql_2.txt`	1.2 Escaping and Hashing
`sql_2-src/`	1.2 Escaping and Hashing
`sql_3.txt`⋆	1.3 The SQL [Extra credit]

Part 2: XSS

Text files, each containing a URL that, when loaded in a browser, immediately carries out the specified XSS attack. For 2.0, also submit the human-readable (non–URL-encoded) payload. Please remember to correctly specify the XSS defense level in your URLs. (If you have nested URLs, specify the defense levels in the nested ones too!)

`xss_payload.html`	2.0 No defenses
`xss_0.txt`	2.0 No defenses
`xss_1.txt`	2.1 Remove “script”
`xss_2.txt`	2.2 Remove several tags
`xss_3.txt`	2.3 Remove some punctuation
`xss_4.txt`⋆	2.4 Encode < and > [Extra credit]

Part 3: CSRF

HTML files that, when loaded in a browser, immediately carry out the specified CSRF attack. Please remember to correctly specify the CSRF defense level in your URLs.

`csrf_0.html`	3.0 No defenses
`csrf_1.html`	3.1 Token validation
`csrf_2.html`⋆	3.2 Token validation, without XSS [Extra credit]

Part 4: Writeup

One PDF file named `writeup.pdf` containing an answer to the questions in part 4, **submitted to Gradescope.**

⋆ These files are optional extra credit.