# Project 2: Web Security

This project is due on **Wednesday, February 15** at **6 p.m.** and counts for 8% of your course grade. Late submissions will be penalized by 10% plus an additional 10% every 5 hours until received. Late work will not be accepted after 19.5 hours past the deadline. If you have a conflict due to travel, interviews, etc., please plan accordingly and turn in your project early.

This is a group project; you will work in **teams of two** and submit one project per team. Please find a partner as soon as possible. If have trouble forming a team, post to Piazza's partner search forum. The final exam will cover project material, so you and your partner should collaborate on each part.

The code and other answers your group submits must be entirely your own work, and you are bound by the Honor Code. You may consult with other students about 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 outside your group. You may consult published references, provided that you appropriately cite them (e.g., with program comments), as you would in an academic paper.

Solutions must be submitted electronically via Canvas, following 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: cross-site scripting (XSS), cross-site request forgery (CSRF), and SQL injection. 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.

# 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://eecs388.org/project2/.

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 download and inspect the Python source code at https://eecs388.org/!/static/project2/bungle-src.tar.gz, 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.

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 never use an important password to test an insecure site!

# Guidelines

**Virtual Machine** For this project, we'd like you to do all your testing within a provided VM, which can be found at https://eecs388.org/!/dist/388-proj2-vm.ova. **The password for this VM is** `project2`. If you do not already have VirtualBox, please download it here https://www.virtualbox.org/wiki/Downloads and import the provided machine. This VM provides Firefox 41 (the browser we will use for grading) and a method for hosting a local HTTPS server (necessary for Part 3). Browser versions have slight variations in their behavior that may affect your XSS and CSRF attacks, so we highly recommend that you develop and test your solutions within the VM.

**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. The solutions you submit must override these selections by including the `csrfdefense=`$n$ or `xssdefense=`$n$ parameter in the target URL, as specified in each task below. **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!

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 web developer tools will be very helpful for this project, particularly the JavaScript console and debugger, DOM inspector, and network monitor. To open these tools, click the Developer button in the Firefox menu. See https://developer.mozilla.org/en-US/docs/Tools. Note that there is also a special version of the browser called Firefox Developer Edition; you're welcome to use it to develop and test, but we will be grading with the normal edition of Firefox.

Although general purpose tools are permitted, your 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 | http://sqlzoo.net/wiki/SQL_Tutorial |
| SQL Statement Syntax | https://dev.mysql.com/doc/refman/5.6/en/sql-syntax.html |
| Introduction to HTML | https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Introduction |
| JavaScript 101 | http://webdesignfromscratch.com/html-css/js101/ |
| Using jQuery Core | https://learn.jquery.com/using-jquery-core/ |
| jQuery API Reference | https://api.jquery.com |
| HTTP Made Really Easy | http://www.jmarshall.com/easy/http/ |

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

# 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://eecs388.org/project2/sqlinject0/`
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://eecs388.org/project2/sqlinject1/`
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 C.

Target: `https://eecs388.org/project2/sqlinject2/`
Submissions: `sql_2.txt` and `sql_2-src.tar.gz`

### 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://eecs388.org/project2/sqlinject3/`
Submission: `sql_3.txt`

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

*URL*
*URL*
*URL*
...

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, as a gzipped tar file (`sql_2-src.tar.gz`). For 1.3, submit a text file as specified.

# Part 2. Cross-site Request Forgery (CSRF)

Your next task 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`".

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.)

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

**2.1 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 3 to accomplish your goal.

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

**2.2 Token validation, without XSS [Extra credit]**
Accomplish the same task as in 2.1 without using XSS.

Target: `https://eecs388.org/project2/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 and load jQuery from the URL http://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js. It is also permissible to load the jQuery cookies library from https://cdnjs.cloudflare.com/ajax/libs/jquery-cookie/1.4.1/jquery.cookie.min.js. Otherwise, your code must be self contained. Test your solutions by opening them as local files in the browser in the provided VM.

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 3. Cross-site Scripting (XSS)

Your final goal 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.

## 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:
`https://localhost:31337/stolen?user=`*`username`*`&last_search=`*`last_search`*

You can test receiving this data within the provided VM by running this command at the shell:

```
$ cd /Documents; python run_server.py
```

and observing the HTTPS GET request that your payload generates. To further test that your HTTPS server is running, try navigating to https://localhost:31337.

## 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.

**3.0 No defenses**

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

For 3.0 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`.

**3.1 Remove "script"**

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

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

### 3.2 Remove several tags

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

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

### 3.3 Remove some punctuation

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

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

### 3.4 Encode < and > [Extra credit]

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

Target: `https://eecs388.org/project2/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 and load jQuery from the URL http://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js. It is also permissible to load the jQuery cookies library from https://cdnjs.cloudflare.com/ajax/libs/jquery-cookie/1.4.1/jquery.cookie.min.js. Otherwise, your code must be self contained. Test your solutions with the browser in the provided VM.

# Part 4. Writeup: Better Defenses

For each of the three kinds of attacks (SQL injection, CSRF, and XSS), write a paragraph of advice for the **Bungle!** developers about what techniques they should use to defend themselves. Place these paragraphs in a file entitled "`writeup.txt`".

**What to submit** A text file named `writeup.txt` containing your security recommendations.

# Submission Checklist

Upload to Canvas a gzipped tar file (`.tar.gz`) named `project2.`*`uniqname1`*`.`*`uniqname2`*`.tar.gz` that contains only the files listed below. These will be autograded, so make sure you have the proper filenames and behaviors. You can generate the tarball at the shell using this command:

```
tar -zcf project2.uniqname1.uniqname2.tar.gz sql_[0123].txt \
  sql_2-src.tar.gz csrf_[012].html xss_payload.html xss_[01234].txt writeup.txt
```

Where applicable, your solutions may contain embedded JavaScript. They are allowed to load **only** the exact Javascript libraries specified above, and must be otherwise self-contained. Test your solutions with the browser provided in the VM, which we will use for grading.

The solutions you submit for parts 2 and 3 **must** include the `csrfdefense=`*`n`* or `xssdefense=`*`n`* parameter in the target URL, as specified in each task. 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.tar.gz` | 1.2 Escaping and Hashing |
| `sql_3.txt`$\star$ | 1.3 The SQL [Extra credit] |

## Part 2: 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` | 2.0 No defenses |
| `csrf_1.html` | 2.1 Token validation |
| `csrf_2.html`$\star$ | 2.2 Token validation, without XSS [Extra credit] |

## Part 3: XSS

Text files, each containing a URL that, when loaded in a browser, immediately carries out the specified XSS attack. For 3.0, 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` | 3.0 | No defenses |
| `xss_0.txt` | 3.0 | No defenses |
| `xss_1.txt` | 3.1 | Remove “script” |
| `xss_2.txt` | 3.2 | Remove several tags |
| `xss_3.txt` | 3.3 | Remove some punctuation |
| `xss_4.txt`$\star$ | 3.4 | Encode < and > [Extra credit] |

## Part 4: Writeup

One text file named `writeup.txt` containing an answer to the questions in part 4.

$\star$ These files are optional extra credit.