Web Security[1]

This project is due on **Monday, April 9** at **11:59 p.m.**. **You may work in teams of THREE if you would like** and submit one project per team (this is strongly suggested).

The code and other answers your group submits must be entirely your own work. 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 Blackboard, following the submission checklist at the end of this file.

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!** You must respect the privacy and property rights of others at all times.

[1] This project is taken from a project designed by Nadia Heninger for her CIS 331 course at UPenn, and I am very grateful to her for letting me borrow it.

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 526, so the investors have hired you to perform a security evaluation before it goes live.

BUNGLE! is available for you to test at http://cs526-s18.cs.purdue.edu/project3/.

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://www.cs.purdue.edu/homes/clg/CS526/projects/proj3.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!

General Guidelines

We have tested this project in Chrome 40.0+. Chrome is available from https://www.google.com/intl/en/chrome/browser/. Many browsers include different client-side defenses against XSS and CSRF that will interfere with your testing. Our web server sets the `X-XSS-Protection` HTTP header to `0` to disable most of these protections. Since we know that Chrome respects this header, we strongly recommend using the most recent version of Chrome for this project (62.0.3202.62 as of this writing); however, if your solution works in a browser other than Chrome, please note this in your submission.

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. And please do not submit the same answer/exploit for two different questions either. Also, you do not need to try to combine the vulnerabilities, except where explicitly stated below.

The extra credit questions are intended to make you think hard. At least one has a clever but fairly straightforward solution, and at least one would require finding a 0-day vuln, as far as we know. Each extra credit question will be worth two points, with an additional bonus two points if you are able to correctly answer them all. This gives a total of ten possible extra credit points for the project.

Resources

Your browser's Developer Tools will be a tremendous help for this project, particularly the JavaScript console and debugger, DOM inspector, and network monitor. The developer tools can be found under View > Developer in Chrome. See https://developer.chrome.com/devtools.

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://sqlzoo.net/
Introduction to HTML	https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Introduction
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:

http://sqlzoo.net/hack/
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: `http://cs526-s18.cs.purdue.edu/project3/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: `http://cs526-s18.cs.purdue.edu/project3/sqlinject1/`
Submission: `sql_1.txt`

1.2 Escaping and hashing [Extra credit]

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";
    }
}
```

You will need to write a program to produce a working exploit. You can use any language you like, but we recommend C.

Target: `http://cs526-s18.cs.purdue.edu/project3/sqlinject2/`
Submissions: `sql_2.txt` and `sql_2.tar.gz`

1.3 Double escaping [Extra credit]

The server works similarly to the previous example, but instead of hashing, it escapes the password in the same way as the username, by calling `mysql_real_escape_string()`.

Target: `http://cs526-s18.cs.purdue.edu/project3/sqlinject3/`
Submission: `sql_3.txt`

What to submit 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.

Part 2. Cross-site Scripting (XSS)

Now that you're warmed up, your 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.)
- OPTIONAL (though useful and suggested for fun/learning): 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 running a basic Python server:

  (Python 2) `$ python -m SimpleHTTPServer 31337`
  (Python 3) `$ python -m http.server 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 four 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: `http://cs526-s18.cs.purdue.edu/project3/search?xssdefense=0`
Submission: `xss_0.txt`

Also submit a human readable version of the code you use to generate your URL, as a file named `xss_payload.html`.

2.1 Remove "script"

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

Target: `http://cs526-s18.cs.purdue.edu/project3/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: `http://cs526-s18.cs.purdue.edu/project3/search?xssdefense=2`
Submission: `xss_2.txt`

2.3 Remove some punctuation

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

Target: `http://cs526-s18.cs.purdue.edu/project3/search?xssdefense=3`
Submission: `xss_3.txt`

2.4 Encode < and > [Extra credit]

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

Target: `http://cs526-s18.cs.purdue.edu/project3/search?xssdefense=4`
Submission: `xss_4.txt`

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 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 Chrome, the browser we will use for grading.

Framework Code

You may build your solution from 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 proxy(href) {
        $("html").load(href, function(){
            $("html").show();
            $("#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://cs526-s18.cs.purdue.edu/project3/";
var attacker = "http://127.0.0.1:31337/";

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

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

Part 3. Cross-site Request Forgery (CSRF)

Your final 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 "`123456`".[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.)

3.0 No defenses

Target: `http://cs526-s18.cs.purdue.edu/project3/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 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 from Part 2 to accomplish your goal.

Target: `http://cs526-s18.cs.purdue.edu/project3/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://cs526-s18.cs.purdue.edu/project3/login?csrfdefense=1&xssdefense=4`
Submission: `csrf_2.html`

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 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 Chrome. 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] The most common password since 2013, when it overtook "password".

Part 4. Writeup: Better Defenses

For each of the three attacks (SQL injection, CSRF, and XSS), write a **short paragraph** about the techniques **BUNGLE!** should use to defend against that attack. Place these paragraphs in a file entitled "`writeup.txt`". If you find any additional security vulnerabilities in the site or have suggestions about how we might improve this project in the future, include them as well.

```
1. [ SQL Injection prevention ... ]

2. [ XSS prevention ... ]

3. [ CSRF prevention ... ]
```

Submission Checklist

Upload to Blackboard a gzipped tarball (`.tar.gz`) named `project3.`*`purdueid1`*`.`*`purdueid2`*`.`*`purdueid3`*`.tar.gz`. The tarball should contain only the files below. When applicable, your solutions may contain embedded JavaScript or CSS, and they may load jQuery from the URL http://ajax.googleapis.com/ajax/libs/jquery/2.0.3/jquery.min.js, but they must be otherwise self-contained. Please make sure you test your solutions in Chrome, the browser we will use for grading.

Part 0: Errata

Please let us know the exact version of Chrome you used for development and testing. If you discovered during testing that your solution does *not* work Chrome, let us know that too.

`browser.txt`

Part 1: SQL Injection

Text files that contain URL-encoded versions of your form fields for the specified SQL injection attacks. These strings will be provided to you by the server when your exploit works.

`sql_0.txt`	1.0	No defenses
`sql_1.txt`	1.1	Simple escaping
`sql_2.txt*`	1.2	Escaping and hashing [Extra credit]
`sql_2.tar.gz*`		
`sql_3.txt*`	1.3	[Extra credit]

Part 2: XSS

Text files, each containing a URL that, when loaded in a browser, immediately carries out the specified XSS attack against the specified target. Also submit an HTML file containing the human readable code you used to generate the URL for part 3.0.

`xss_payload.html`		
`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 against the specified target.

`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

A text file containing your suggested defenses against each of the attacks.

`writeup.txt`