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

BUNGLE! is available for you to test at **http://cs558web.bu.edu/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 http://www.cs.bu.edu/~goldbe/teaching/HW55814/lab3/web.rar, 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!

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 the necessary HTML files 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. [12 points]
Target: `/login` with `csrfdefense=0` and `xssdefense=4`
Submission: `csrf_0.html`

2.1 Token validation. [20 points]
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. This cookie is not session specific but it remains the same for a given IP, a given browser and a given time window.

To side-step this defense mechanism, we will be using the XSS attack from part 3. Go ahead and do 3.0 first, before proceeding with this part. You need to construct now a single HTML file that upon execution, first hijacks the cookie and then proceeds to log-in as "`attacker`" achieving the same effect as in part 2.0 above.

Target: `/login` with `csrfdefense=1` and `xssdefense=0`
Submission: `csrf_1.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 or a newer version of jQuery. **Make sure you test your solutions by opening them as local files in Firefox. We will use this setup for grading.**

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)

In this section we will demonstrate a "light" 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 settings below, your goal is to construct an HTML that upon execution, correctly executes the attack specified.

3.0 No defenses. [6 points]
Start with a basic attack, where the payload simply produces an alert box that contains the cookie described in section 2.1 above.
Target: `/search?xssdefense=0`
Submission: `xss_0.html`

3.1 Report cookie. [14 points]
Clearly, the attack above is very limited. An attacker that convinces the victim to run the link does not receive the cookie. For this part, we will strengthen the attack by having it report the victim's cookie back to the attacker. In a real world scenario this report would be received by the attacker's server; for the needs of this lab, attacker and victim are "sitting" at the same machine, hence we will have the attack report the cookie to a"virtual" side channel. Namely, cookie should be reported at the localhost (IP address 127.0.0.1) at port 31337.

In order to capture incoming traffic at port 31337, use Netcat[1] by running `$ nc -l 31337` (code **Hint** at the bonus part below will help you see how to send info back to the "attacker"). Upon execution of the supplied HTML file, an attacker listening to port 31337 should be able to read the csrf token (possibly among other things).
Target: `/search?xssdefense=0`
Submission: `xss_1.html`

3.2 Remove "script". [12 points]
Repeat part 3.1 above but this time the server deploys a defense by removing all occurences of "script" from the submitted search query, using the following code:

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

Target: `/search?xssdefense=1`
Submission: `xss_2.html`

3.3 Remove several tags [12 points]
Repeat part 3.1 above but this time the server deploys a defense by removing the following tags:

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

Target: `/search?xssdefense=2`
Submission: `xss_3.txt`

[1] On Windows machines download and install Nmap from here http://nmap.org/download.html. Then from terminal navigate to the folder where you installed it and run `$ ncat -vv -k -l 31337`.