|  |
| --- |
| **CSE 3140 Lab 6: Cross-Site Web Attacks**  **Created: Fall 2022 by Amir Herzberg, Updated: Spring 2023 by Amir Herzberg**  **Monday sections submit by 12/10**  **Wednesday sections submit by 12/5** |

Web applications are critical component of our lives, society, and economy. Consequently, their security is important – but also, as we will see, challenging. In this lab, we will learn some basic topics of web security, focusing on (security against) attacks by **rogue websites,** including two of the most important attacks: **Cross-site scripting (XSS)** and **Cross-Site Request Forgery (CSRF),** and basic defenses **filtering** and **tokens.** We will also learn about [**cookies**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cookie)**,** an essential component of web programming that we didn’t cover yet, which is important for both security and privacy. Both attacks we will see are related to cookies.

You can learn more about web security and privacy in CSE 4402 – and from many excellent online sources (websites) and books.

**Question 1A (5 points)**

The communication between browsers and webservers uses the HTTP protocol, where the browser sends *HTTP requests* and the server responds with *HTTP responses*. The HTTP protocol is *stateless*, i.e., the server handles each request separately, without remembering the previous interaction (requests, responses) from the same user. So how can the server identify that a request comes from a specific user, and that the user provided correct password, without having the user provide these values again and again in each page? This is achieved by a mechanism called [*cookies*](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cookie).

A cookie is a (name, value) pair which is sent by server in a *Set-Cookie* header, as part of an HTTP response. The browser stores the cookies it receives from webservers. Later, whenever the browser sends another HTTP request to the same server, it includes with the request all of the cookies that the browser received from the same server. As you rightfully suspected, this description is a bit simplified; and anyway you may want to see a [well-written and clear explanation of cookies and the relevant headers, e.g., from the MDN site](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cookie). The application on the web-server receives the cookie when send by the client, and can then identify different requests from the same browser, e.g., to know when receiving requests from a currently logged-in user. The server can also use the cookie as a key to lookup state information stored in files/DB on the server. That’s how websites work, without requiring re-entry of username / password on every page. Cookies also allow websites to provide convenient services such as shopping-carts and annoying things like advertisements, without requiring users to login.

Note: cookies are also related to **privacy**, and there have been recent changes in the Cookies mechanisms related to privacy. You can read about this online or in CSE 4402 (web+net security).

In the file *Q1login* in the *Lab6* folder of your VM, you will find *two* (username, password) pairs (for each of you); one begins with the letter A (for *attacker*) and the other with the letter B (for *benign user*).

Open our `bank’ site (<http://bank.com>), and login to the ‘B’ user. You will be now `logged in’ and be able to transfer money to other accounts; this is facilitated by a cookie sent by the server to your browser. *Find the cookie(s)* that your browser received from bank.com. This information can be found directly from your browser, using its user interface; the method is browser-specific but not hard to find.

**Submit in submit.edu:** the value of the cookie you found. **Submit in your lab report document in HuskyCT:** the pair of usernames you picked (‘A’ and ‘B’), and the details of the cookie you received after login to the `B’ user; include and explain these details in text and include a screen shot of the browser dialog which provided this information.

**Question 1B (15 points)**

In this question, you will write a simple Flask website using cookies to identify returning users. The website will set the cookie by sending it to the browser, as part of the server’s **response**, in the **Set-Cookie** http header. When the browser sends a **request** to the server, it automatically attaches the cookie(s), using the aptly named **Cookie** http header. It is not difficult for your Flask website to use cookies (i.e., set them in responses sent by the server and use them in requests received by the server); you can learn about this easily, e.g., simply from the [Flask quickstart](https://flask.palletsprojects.com/en/2.2.x/quickstart/). And there are plenty of resources if you want to learn more about [HTTP requests and responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages) and [cookies](https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies); in fact, here’s [another site on cookies,](https://www.w3schools.com/js/js_cookies.asp) discussing how you can create, read and delete them with JavaScript. That’s cool (and relevant to this lab).

Your flask server should setup the following cookies:

1. Cookie Q1B1, whose value is your netid (repeat for both students in the pair).
2. Cookie Q1B2, whose value is your last name. Setup this cookie so that it would be sent only for requests to folder Q1B2 of your website.
3. Cookie Q1B3, whose value is the IP address of your VM. Setup this cookie to prevent exposure to a rogue cross-site script.

**Submit in your lab report in huskyCT:** screen capture (recording) showing how you identify repeating visits of different users using the cookies, and showing the cookies using the browser’s UI; your website scripts; the confirmation code you received; and explanation of your scripts and cookie options you used.

**Question 2 (20 points)**

In this lab, we will discuss two of the most well-known (and widely exploited) attacks by rogue websites: the *Cross-Site Scripting (XSS)* attack and the *Cross-Site Request Forgery (CSRF)* attack. As you noticed, both are `cross-site’ attacks; this means that they are attacks by *a rogue website*, against a user which is using both the rogue site and another, `victim’, website, not necessarily at the same time. (We simplify a bit; to learn more, take CSE 4402… or learn by yourself.) In both attacks, the attack involves attacking Javascript code, which was written by the attacker and is running in the user’s browser. You may want to take a look at the [Flask security considerations](https://flask.palletsprojects.com/en/2.2.x/security/) page, which covers, briefly, these and other security aspects, some of which relevant to this lab.

We begin with the simpler attack: **Cross-Site Request Forgery (CSRF).** In the CSRF attack, a rogue website, visited by the victim user, embeds an HTML tag (such as <IMG>, <SRC>, etc.) that will invoke a request to the victim’s webpage. Normally, such requests are used to retrieve a specific object kept by the server, and embed that object in the page displayed by the browser. For example, the <IMG> tag embeds an image, and the <SRC> tag embeds a script (written in JavaScript). However, in a CSRF attack, the request would not be to any object kept by the server; instead, it would be formatted exactly as a request to a specific *action* to be done by the server; an action that *should* be authorized by the user, such as transfer of funds from the user’s account to the attacker’s account. The server authenticates the user using a cookie sent by the browser with the request; however, while this authenticates correctly the current user of the browser, this does not mean that the user has intentionally sent the request – indeed, in the CSRF attack, the request is made by another webpage visited by the user. This is exactly what you would do in this question!

Your first step may be to identify the exact format of funds-transfer requests performed when a user fills the `funds transfer’ form in the `Husky Banking’ website (after login). One way to do this is using the [browser’s developer console/tools](https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_are_browser_developer_tools); another would be do look in the `source’ (HTML) of the (simple) Husky-Banking page containing the funds transfer form. Anyway, in this or another way, find the format. We recommend you try it, manually, but with a very small transfer (e.g., $1); if you simply enter in the browser `location’ bar the corresponding URL, the transfer should succeed. This URL would be something like [http://bank.com/loggedIn?username=Benign&moneyAmount=5](http://bank.com/loggedIn), where Benign is the user who is receiving funds, and 5 being the amount they are receiving.

Next, make the `attacker’s website’ using a tag (e.g., <IMG>) that will cause the browser to issue a request to our Husky Banking server, causing transfer of the amount in file Q2 from your `B’ account to your `A’ account. You can test it by visiting your own website and checking for the move of funds from the `B' account to the `A’ account.

Once your website works, **show the attack to the TA** who would give you a confirmation code, which you should **submit** in *submit.edu.*

**Submit in your lab report in huskyCT:** the confirmation code, all the code of your website (as text), screen capture (recording) showing how you tested your attack website; and explanation of your scripts and of how the attack works.

Note: due to privacy and security concerns, modern browsers restrict sending of cookies in requests sent from other websites; you can read about the **SameSite** attribute, which is the main way to restrict such sending of cookies. This mechanism that can foil this attack.

**Question 3 (10 points)**

In this question we move to an **XSS attack.** A successful XSS attack allows the attackers to run their malicious JavaScript, as if it is a part of the victim’s webpage. XSS attacks exploit the fact that the HTML from the server contains *code/control* information – tags and JavaScript – as well as *data / text* (e.g., the user’s posts). The code/control information is marked by *HTML tags* - but these tags are also textual. What happens if the HTML tags may appear, by chance or intentionally, within text, e.g., as part of a comment posted by the user? Well, the browser will *execute* it! In fact, there are other situations where the browser may run JavaScript code, from a location in the webpage where the designer expected to have non-executable information; for example, if JavaScript takes the place of as URL being linked to from the page (using the <a href=….> HTML tag). Basically, XSS is due to the fact that in scripting languages, like JavaScript (and Python!), **there is no clean separation between the code/control and the data/text. Separation is important for security.**

XSS is a very common and critical attack. Therefore, any serious web framework, including Flask, has, by default, defenses against it; indeed, we took special care to make our Flask `Husky Banking’ site vulnerable to XSS attacks, including the very basic XSS attack that we use in this lab.

In this question, we simply ask you to perform a simple test to identify simple XSS vulnerability in our Husky Banking site. In this test, you try to enter, `manually’, a trivially observable JavaScript code in different input fields in the Husky Banking website. For example, we can try to enter a string containing the simple JavaScript code: <script>alert("Site is vulnerable to XSS!")</script> ; if executed, this script would display an ‘alert’ on the browser’s window, with the message ‘Site is vulnerable to XSS!’. For this attack to work, the website would ‘echo’ this text, containing the script, in the HTTP response to the request containing this text.

Once you find a field which is vulnerable to XSS, it is time to exploit it. In this question, we see the simplest exploit (and we also do it manually). You want to use an <iframe> tag within your website, where the source url has the query variables filled with malicious script that can display the users cookie.

Once your website works, **show the attack to the TA** who would give you a confirmation code, which you should **submit** in *submit.edu.*

**Submit in submit.edu:** the value of the cookie and the confirmation code from the TA.

**Submit in your lab report in HuskyCT:** the *value* of the cookie and the confirmation code from submit.edu, and a screen capture (recording) showing your testing of the fields of the Husky Banking site, including the display of the cookie.

**Question 4 (25 points)**

In this question, you write an XSS-attack site. To make it more interesting, this site will find a *new* cookie, not the one you found in the previous question. At the Husky Banking site page Q4, there is some input fields that allow you to make transactions between users. The input fields usually sanitize what is given to them, rejecting anything that is not an integer value. However, we intentionally included a mechanism that disables the sanitation, which allows the attack; such mechanisms that disable defenses are referred to as *trapdoors* or *backdoor*; they are often installed and used by rogue programmers, for nefarious purposes, but also used by benign programmers for educational purposes (like we do) or to debug and test programs. They are a special case of *easter eggs,* which is the name for different undocumented software functionalities, often installed for fun (e.g., in games).

Anyway, you will need to find the backdoor that we installed, which disables the sanitation of the inputs, and would therefore allow you to perform the XSS attack against the Q4 page. The backdoor is simply a magic number that is between 0-1,000, and given at the beginning of the vulnerable input field. If the correct magic number is used, then all subsequent text after that integer is not sanitized. You will need to write a script that will find this magic number (by trying all numbers between 0 and 1000). After finding the magic number, you will be able to run the XSS attack against the Q4 page and find the corresponding cookie from your attacking website

Once your website works, **show the attack to the TA** who would give you a confirmation code, which you should **submit** in *submit.edu.*

**Submit in submit.edu:** the value of the cookie and the confirmation code from the TA.

**Submit in your lab report in HuskyCT:** the *value* of the cookie and the confirmation code from submit.edu, all code of your website (as text in the report), and a screen capture (recording) showing your testing of your website, including the display of the cookie.

**Question 5 (25 points)**

Stealing cookies using XSS attacks, as in the previous question, is so common, that there is an optional flag for cookies called *HttpOnly*. A cookie for which this flag is set, is not accessible at all by JavaScript, and only sent to the server.

In this question we do a different XSS attack, called **web-page defacement,** which does not require the rogue script to access a cookie**.** Instead, the rogue script, when executed by the browser as result of XSS, will **change the contents of the webpage as displayed to the user and used by the browser.** Webpage defacement is often used to display a political message, as part of *hacktivism* or influence public opinion or voting; or, webpage defacement may be used to try to `trick’ the users into downloading malware or into exposing their credentials.

In this question, your goal is to trick users that they have made a transfer of $666,666 and that they should call 666-390-6590 for help. This defacement can be done by inserting an image of this transfer on your site or modifying the received HTML through DOM.

In addition, embed in the page a `defacement image’, which should be retrieved by the browser from bank.com/cat.jpeg or your own (attacker) location.

Once your website works, **show the attack to the TA** who would give you a confirmation code, which you should **submit** in *submit.edu.*

**Submit in your lab report in huskyCT:** the confirmation code, all the code of your website (as text), screen capture (recording) showing how you tested your attack website; and explanation of your scripts and of how the attack works.