CSE 3140 Lab 6: Cross-Site Web Attacks

Created: Fall 2022 by Amir Herzberg, Updated: Spring 2024

Monday sections submit by 4/28

Wednesday sections submit by 4/30

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,** 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.

Throughout this lab you'll need to access the same banking website as in Lab 4. As a reminder, this website is located at 172.16.48.80. You need to use SSH tunneling as described in Lab 0 to access.

QUESTION 1A (10 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 on each page? This is achieved by a mechanism called *cookies*.

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 web servers. Later, whenever the browser sends another HTTP request to the same server, it includes with the request all 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. The application on the web server receives the cookie after it is sent by the client and can then identify different requests from the same browser (e.g., it can know when it is 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/databases on the server. That's how websites can 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 (both meant for you to use); 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. For our Husky Banking website, the cookie used to maintain a "logged in" state is the LOGIN_INFO cookie.

Submit to the autograder: the value of the LOGIN_INFO 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. And there are plenty of resources if you want to learn more about HTTP requests and responses and cookies; in fact, here's another site on cookies, discussing how you can create, read and delete them with JavaScript, and you can learn about restricting access to cookies here. That's all 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. For this, look into the Path attribute of cookies.
3. Cookie Q1B3, whose value is the IP address of your VM. Setup this cookie to prevent exposure to a rogue cross-site script. For this, take a look at the section on Security in the MDN cookies documentation for cookie attributes that you can set to help with this task.

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; and an explanation of your scripts and cookie options you used.

QUESTION 2 (25 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 who 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 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 for 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 correctly authenticates the current user of the browser, this does not mean that the user has intentionally sent the request. In the CSRF attack, the request is made by another webpage visited by the user.

1. 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; another would be do look in the `source'

(HTML) of the (simple) Husky-Banking page containing the funds transfer form. We recommend you manually try it with a very small transfer (e.g., $1). You will know you understand the format when you can enter in the browser `location' bar the corresponding URL and have the transfer succeed. This URL would be something like http://bank.com/loggedIn?username=Benign&moneyAmount=5, where Benign is the user who is receiving funds, and 5 being the amount they are receiving.

2. 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.
3. Once your website works, **record an example of the attack showing the transfer of funds** to include in your lab report.

Submit in your lab report on HuskyCT: 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 , e.g. here, about the SameSite cookie attribute (and other attributes). A cookie with SameSite set to the default value SameSite=Lax would not be sent in a cross-site request such as generated by the <IMG> or <IFRAME> tags, which would foil this CSRF attack (that was the main motivation for introducing SameSite). To allow your attack to work, we intentionally disabled this mechanism by setting SameSite=None in the cookies of our Husky banking website.

QUESTION 3 (20 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.

If the HTML tags appear, by chance or intentionally, within the text, e.g., as part of a comment posted by the user **the browser will *execute* it!**

There are situations where the website designer expected there to be non-executable information. However, the browser will run any code it finds (properly formatted). For example, if a fixed URL linked in a page is replaced by JavaScript (using the <a href=….> HTML tag). The source of XSS is that in scripting languages, including JavaScript and Python, **there is no separation between the code/control and the data/text. Separation is important for security.**

XSS is a common and critical attack. It is frequently in the OWASP top 10. Web frameworks, including Flask, have defenses. We made our Flask `Husky Banking' site vulnerable to XSS attacks, including the very basic XSS attack that we use in this lab.

You need to identify XSS vulnerabilities in our Husky Banking site. To find such XSS vulnerability, we recommend that you enter 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 successful, this script would display an 'alert' on the browser's window, with the message 'Site is vulnerable to XSS!'. At a technical level, the website echoes this text, containing the script, in the HTTP response to the request containing this text.

You can then repeat the above process but using a slightly different script, that will display, in the alert window, the value of the cookie.

Finally, now that we've found and explored this vulnerability in the Husky Banking website, we are ready to exploit it using our attack website.

Adapt your attack website (from Q2), to create a reflection XSS attack, where the script is sent in a request to our Husky Banking site that is generated by your attack. This will cause the Husky Banking website to run the script (and show the cookie on the browser's window). You may want to use an <iframe> tag within your attack website, where the source URL points to the Husky Banking website and has the query variables filled with malicious script that displays the user's cookie.

Once your website works, **record an example of the attack** to include alongside your lab report.

Submit to the autograder: the value from the Q3 cookie.

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

QUESTION 4 (30 POINTS)

In this question, you will write a XSS-attack site, finding a new cookie. At the Husky Banking site page Q4, some input fields allow you to make transactions between users. A good input fields will sanitize what is given to them. For example, an amount field might reject something that isn't an integer. For this question, we disabled sanitation. Such disabling is done for malicious intent (called a trapdoor or backdoor), educational intent (us), fun intent (called easter eggs), or by careless or clueless programmers.

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

a magic number that is between 0-1,000. When the correct number is given at the beginning of the vulnerable input field, then all subsequent text after that integer is not sanitized.

Write a script that finds 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, **record an example of the attack** to include alongside your lab report.

Submit to the autograder: the value of the magicCookie cookie.

Submit in your lab report on HuskyCT: the *value* of the magicCookie cookie, 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.