Cybersecurity Lab, CSE 3140
Spring 2022

Web Application Security

Section #: 4
Team #: 13
Names:
Hunter Krasnicki, Jonathan Ameri

Web applications are a 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, including two of the most important attacks: **Cross-site scripting (XSS)** and **Cross-Site Request Forgery (CSRF),** and basic defenses: **cookies, filtering,** and **tokens.** You can learn more about web security in CSE 4402. We will also learn some basic web technologies, mainly: HTML, HTTP, PHP and JavaScript.

Please backup your page and files for this question (and every question), don't corrupt what you did!

Tools used

Web Browser. This lab may be done from home by VPN access to your VM in the lab. In the lab you would write a website, using the Apache webserver installed on your VM. You will access the site from the browser using the IP-address is of your VM. Each group is assigned two VMs using the IP 172.16.50.X and 172.16.51.X (X= 20*section number + group number)

Question 1 (5 points)

In this question, you will write a simple website, using HTML, with a simple `Hello World' script. The script will be in JavaScript; the goal of this question is to give you basic experience in HTML and JavaScript. Both are widely used and widely documented languages; here are some good introductions:

- HTML: w3schools, Mozilla
- JavaScript: w3schools, Mozilla

Of course, there is a lot more you can learn about web technologies and security, from these sites and elsewhere… but let's stay focused on the lab, Ok?

Your web page should:

- Run on the webserver and accessed by the browser. If you like, you can first experiment by writing this site as a file on your local computer, as it does not require any webserver support.
- Display the name of the lab, section, group, and names of the team
- Include a script that runs – automatically and/or upon pressing a button – and display `Hello, world – it's (section – group – names)'.

Submit: webpage (source) and screen shot. Some of you may already know HTML and JavaScript; in this case, make a nice website, also using CSS (Cascading Style Sheet) to control its appearance. But if you had to learn HTML and JavaScript, you do not have to make it pretty or to learn and use CSS. You'll learn enough without these!

Question 2 (10 points)

Let us now make the site a bit more interesting, using some simple HTML and webserver (PHP) functions. You may want to read a bit about PHP (Click on the links).

Specifically:

1. Add a bit of text at the top of the page, including some **bold text,** some *italics* text, some bullets, a title identifying the page, and some other cute text features. *It'll be nice if your text briefly describes your experience with the lab!*
2. Add a **form** to the page, allowing the user to submit a comment. The server will then display the comment (separately, marked as a comment, at the bottom of the page). When you revisit the page, it should automatically display the comment, until the visitor enters a new comment. Then you can replace the old comment with the new (or display both, as you like). Check this link

Submit:

- Screen shots of the page: without comments, with one comment, with a new comment
- Screen shots of the source of the webpage as stored in the server and as received by browser.

Question 3 (10 points)

In this question we further upgrade our website, to allow for multiple users. Namely, add, in the form, a place to enter 'username' and 'password', in addition to the 'comment'. Also add a 'Register' button, which links to a separate registration page that you will also do, where a new user can register to the site (entering name and password), check this link. Your site should store the name-password pairs and verify the password before accepting a comment. When you display a comment, add the name of the user who made that comment!

Submit:

- Screenshots of the registration page and of the `main' page with comments from two users.
- Screenshots of the source of the webpage as stored in the server and as received by browser.

Question 4 (10 points)

In this question we... sure, further improve our site! So, it's a bit annoying that whenever we visit the page and want to make a comment, we need to re-enter our username and password, no? Can't the web-server remember which user it is? We'll add support for this – but it's not automatically enabled by HTTP, since, for simplicity and efficiency, HTTP is **stateless** - the web-server does NOT maintain any state information. But, the **browser** can maintain state information; and your application on the web-server can use information from the browser to lookup state information stored in files/DB on the server. That's how websites work, without requiring entry of user-name/password all the time, and providing

convenient services such as shopping-carts and annoying things like advertisements. (More on that later.)

The mechanism to maintain state on the browser is called **cookies**. The server can set the cookie by sending to the browser, as part of the server's **response**, the **Set-Cookie** header. When the browser sends a **request** to the server, it automatically attaches the cookie(s), using the aptly named **Cookie** header.

This may be a good point for you to learn a bit about the web protocol, HTTP. HTTP is a rather simple protocol, allowing clients to send requests to servers and to server to return responses; both requests and responses are encoded by readable text, and begin with a series of headers followed by optional payload. One good place to read a bit about HTTP, is in the MDN site of Mozilla; see their discussion of HTTP requests and responses and of cookies. Here's another site on cookies, discussing how you can create, read and delete them with JavaScript - and here on handling cookies with PHP.

Please backup page and files for this question (and every question), don't corrupt what you did!

So... Let's get to work! Add to your website another page, for login. And add the cookie mechanism, so that once a user has logged-in, the server sets a cookie; and when the user re-enters the site (with a cookie), the user is identified using the cookie, so **userid/password fields are not even shown.** Include a button to `logout', allowing you to re-login as a different user. a good start can be found here and here

Submit:

- Screenshots showing the improved login process, including changing users, entering wrong passwords, etc.
- Sources of your code.
- Results of Wireshark, showing the relevant HTTP request / response, including the cookies; point out to the cookies!

Question 5 (10 points)

In this question we... finally do an attack. Yes, this website is insecure... (unless you implemented some defenses, we did not tell you to implement!)

Look again in the HTML from the server with your comment (from previous questions). Notice that it contains *code/control* information – tags and **JavaScript** – as well as *data / text* (e.g., your comment). The code/control information is marked by *HTML tags* - but these tags are also textual. The HTML tags may appear, by chance or intentionally, within text, e.g., as part of a comment posted by the user!

This is a serious security problem: t**here is no clean separation between the code/control and the data/text! Separation is important for security; the lack of separation here will be abused by the XSS attack, which we will soon see.**

In this question, you will use the lack of separation, to allow **a rogue user** to `embed' a simple script (like in Question 1) in the webpage returned by the server to a **victim user.** The attacker will embed the script within the comment that the website allows users to make; the server will store the comment including the script and provide it to all users – and the script would then run-in other user's browsers!

First, do a simple experiment: embed some simple tag, e.g., text in the comment. What is the impact?

Next, let's do a simple attack. Write a rogue script that will **change the contents of the webpage displayed to the user.** The contents should be **very different** from the original contents; bonus points will be given to students who will do *interesting* and/or *funny* changes. Be creative!

Upload the rogue user's script to the server, by entering it as a comment by the **rogue user** (aka User1). Observe the script running when you load the page!

Finally, login as a victim user (aka User2) and view the page. Since you should see the comment made by the rogue user, you should also be able to observe the script running. Namely, the rogue user was able to post a comment including a script, and the script is run on the browser of the victim user (User2)!

This is called **Cross-Site Scripting** *or* **XSS** (since the script came `across the site' from User1 to User2). The specific exploit is referred to as **web-page defacement.** Defacement attacks are used in real attacks: to send political and other `messages', to mislead viewers, and more.

Submit:

- Screen shots of all steps, including the final one (user2 visiting the webpage running the script uploaded by user1).
- The HTML of that page from the browser of User2, showing the script.

By causing User2 to run a script uploaded by User 1, we allow User 1 to attack User 2 – this is called a **Stored XSS attack.** It is `stored' since the code is stored by the server. We will turn it into a more `convincing' stored XSS attack in the next question.

Question 6 (10 points):

In the previous question we already did a simple stored XSS attack, i.e., User 1 uploaded a script which is then executed when User 2 accesses the server's webpage. In this question, we will turn this into a more serious stored XSS attack, allowing User 1 to impersonate User 2, by **stealing User1's cookie**.

The attack is based on the fact that JavaScript can access the current cookies; e.g., read here. But notice that a cookie may have a flag called **httponly** which will make it unavailable to scripts on the browsers. So:

- Try the attack and if the script cannot access the cookie due to httponly flag, change your PhP code and/or server configuration to disable httponly.
- And if the script did access the cookie… then change the PhP or configuration now so that it will add the httponly flag – and see how the script now **cannot** access the cookie!
- In the rest of the XSS questions, ensure httponly is NOT used (to make attacks work).
- Note: httponly does not prevent all attacks – e.g., defacement, and even some clever ways to steal cookies… but that's all beyond our lab.

Change your script to display the cookie. **Submit** screen shot(s) showing User 2 accessing the website (with the stored script), and the browser running the script and displaying the cookie.

Of course, the attacker needs to **receive** the cookie, not to just display it to the user… which we do in next question.

Submit screen shots and code, as usual.

Question 7 (10 points)

Now, you will implement the attacker's web server. **Note this website should run at a different IP address (that's why you were given two).**
At this point, the website is just used to collect cookies. Namely, you will modify the attacking script so that it would **send the cookie to the attacker**'s website. The website will simply save these cookies. The attacker's site will include a page displaying collected cookies, with a link allowing the cookie to be added to the browser, and another link connecting to the victim webpage (with the form) - with the chosen cookie...

Submit: the code of the website and screenshots showing all steps of the attack.

Question 8 (10 points)

Ok, by now we've seen two exploits of stored-XSS attacks (defacement and exposing the cookie to allow unauthorized access as a different user). Time to learn a defense... We will now add to the site a simple defense: **input filtering**. Note that this defense is not very strong, e.g., you can find online different tricks that may allow circumventing it (e.g., XSS cheat sheet). We will even see this defense fails to prevent another XSS attack. So serious websites use better defenses. But input filtering will do for this question.

You can read about input filtering in PhP.

Done? Check it – try the attack again. Where does it fail?

Submit: the code of the website and screenshots showing how the attack fails now; explain the difference to the unprotected site.

Question 9 (10 points)

We already said input filtering isn't enough… Let's see one reason, by presenting a classical **reflected XSS attack.** A reflected XSS attack begins when a victim user (our User2) visits the attacker's rogue website. The websites instruct the browser to automatically send a request to another website. This is a special request: (1) the request string **contains a (rogue) script,** and (2) the request somehow causes the server to return the same string to the browser. As a result, the rogue script runs in the victim's browser – and can perform defacement, steal cookies, etc….

The specific reflected-XSS we'll do is a very classical one: the **404 not found XSS.** When you type a URL incorrectly, or even follow an old hyperlink leading to a URL which is not available anymore, you'll get a webpage alerting you to the error, right? Such pages are usually referred to as **404 not found** pages, since HTTP servers automatically send them, with error code 404, when receiving a request to a non-existing page/resource. Furthermore, in many webservers, the response **echos the not-found request** – probably, to make it easier to find out the problem.

However, this mechanism may allow a simple **reflected XSS attack:** the attacker causes the victim user's browser to **send a request for a `resource' – which is actually the rogue script!**

Would this work against your webserver? Let's find out.

1. Test if, upon receiving a request for a non-existing web-page/resource, your web server returns an 404 page echoing the request. (If not: attack will not work; find if you can change the response to the `standard' one which is vulnerable, if not, move to next question!)
2. Test if, when the request contains a script (your `hello world' from Q1 will do), then the response page in the browser would run the script. (If not: attack will not work, see if you can find out why to allow the attack; if not, move to next question!)
3. Create another web-page for the attacker. This webpage should emulate some web page to which we can attract victims – jokes, pictures, whatever. In this webpage, you'll embed an HTML tag that will **automatically** send a request to the `legitimate' website – so that no user involvement would be necessary. For example, you can use the image tag (<img src="xyz.jpg">). The request will contain the attacking XSS script from Q7!

Submit: screenshots showing the steps above and their results, including the attacker's code.

Question 10 (5 points)

One simple way to prevent the 404 Not Found attack is to provide a different 404 response page – which will not be vulnerable, e.g., by using input filtering. Do it; one way is described here.

So, do a new 404 response page which will not allow the XSS attack. Make it give a funny response!

Submit: screen shots showing the new page, and the code of the response page; describe any configuration changes.

Question 11 (5 points)

Let us now briefly see one more common attack: **Cross-Site Request Forgery (CSRF).** In this attack, the rogue website, visited by the victim user, will again embed an HTML tag such as <IMG> that will invoke a call to the victim's webpage. The difference is that this time, we will not rely on the 404 attack – you just fixed it. In fact, we don't do XSS at all. Instead, we simply **send a request** to add **adversary's own comment** to the webpage. In practice, this attack can be used to submit more critical operations, such as moving money to the attacker's account (for a banking website).

The attack works since the browser automatically attaches the cookies to the request. Note: this is also how many websites perform `tracking' to identify users across websites, by including on the website a tag such as <IMG> to a cookie-tracing website, often an advertising service such as **doubleclick** (of Google) or **facebook**.

Due to privacy and security concerns, modern browsers restrict sending cookies from one site to another; you can read about the **samesite** attribute (the main defense). This is one mechanism that can foil this attack; so, if it does not work with your browser, you may want to do the attack intentionally from an older browser such as the IE on your lab laptops. Actually, we believe any version of IE should work.

Submit: screen shots of the attack and the code of the attacker’s webpage.