School of Computing and Information Systems

COMP90074: Web Security

Workshop Week 2

1 Introduction

In this lab we will explore XSS vulnerabilities as demonstrated in a simple Python web server. You will think about exploitation, and look at mitigation strategies that can be deployed.

We will be configuring the Python web server on Docker.

The precise organisation for the Python web servers is still evolving and so may change in future workshops.

2 Prepare the Lab Environment

In this workshop you will be running a web server on your local computer, within a Docker container. You will be interacting with the web server using your own web browser. You will also make changes to the web server's implementation.

We provide a script to automatically set up the environment, though please feel free to edit and try out different settings.

The web server is written in Flask. Flask is a lightweight web framework for Python. It makes creating web applications relatively quick and simple, and takes care of a lot of the underlying implementation details.

1. Make sure you have Docker installed on the device you plan to complete this workshop
2. Download the zip file for this workshop from the LMS. Unpack the zip file. This will create the `workshop2` directory, which contains the web server and a Dockerfile for running it within a docker container.
3. The server can be run by executing from your terminal the `run_server.sh` script. If you can't execute this script (e.g., you are running on Windows without Cygwin), you can instead manually run the following two commands:

   ```
   docker build --tag python-docker .
   docker run -p 80:80 python-docker
   ```

   If running successfully you should see output that ends with something like the following:

   ```
    * Running on all addresses (0.0.0.0)
    * Running on http://127.0.0.1:80
    * Running on http://172.17.0.2:80
   Press CTRL+C to quit
   ```

This means the server is running. You can connect to the web application by visiting "localhost" on your web browser.

Note that the `-p 80:80` option above maps the port 80 (the standard port for HTTP connections) on your computer to port 80 within the Docker container, allowing your web browser to talk to the web server inside the container by visiting "localhost". If your device has the port 80 already occupied, you may try another port, for example, `-p 5000:80` and then visiting "localhost:5000" in your web browser.

4. To stop the server press `CTRL+C`

2.1 The Web Application

The web application initially displays a login page that accepts a username and password. You can log in with any username, and the password "letmein".

This leads to a broadcast message application in which users can message all other logged-in users. Each user sees all messages that have been sent using the application (since it was last re-started). Users can also filter the displayed messages by keyword. Users can also change the look and feel of the application: in this simple example the colour scheme.

Logging in to the application sets a session cookie, allowing the user to interact directly with the application without having to log-in again until they logout.

2.2 Understanding the Impact of each Vulnerability

The web application contains a number of XSS vulnerabilities, as follows:

- the "msg" parameter on the login page (identical to the demo from Lecture 1);
- the filter keyword functionality, via the "filter" parameter on the application page;
- the colour scheme on the application page, which is provided in the URL hash component; and
- stored messages, which have previously been sent by other users (identical to one of the demos in Lecture 2).

1. For each of the first three vulnerabilities, imagine you are an attacker trying to exploit each one of them. You will do so by crafting a link (URL) to the application and then emailing that link to a user of the application. When the user clicks the link in the email, it will cause their browser to load the application (perhaps authenticating with an existing session cookie, because the user had already logged in) and execute malicious JavaScript contained in the link.

   Think about what bad things your JavaScript might be able to do, either to the user or on their behalf. Practice writing example URLs (links) that, for each vulnerability, injects JavaScript into the page. Test out your links by pasting them into your own browser and seeing what happens.

   You may wish to open the developer JavaScript console to help debug your links.

   Note: When testing these XSS attacks your browser or extensions may try to block them. You may need to temporarily disabled web security extensions, or disable built in browser protections. For example, Google Chrome has an XSS auditor that can be disabled by starting it with the following argument `-disable-xss-auditor`. A full list of Chrome flags can be found at `https://peter.sh/experiments/chromium-command-line-switches/` and instructions for using them at `https://www.chromium.org/developers/how-tos/run-chromium-with-flags`

   Note: the goal here is of course not to cause harm but to get you to practice adversarial thinking. The only way to properly understand the potential impact of a vulnerability is to have a good understanding of how it can be exploited.

2. Do the same for the last vulnerability (stored messages). Now, the scenario is that you are an attacker who is using this application. You want to submit a malicious message that will cause JavaScript to be executed by *everyone* who uses the messaging application. Think about what your malicious JavaScript could do that the attacker (who is also a user of this web application) couldn't do already. What would your JavaScript need to do to accomplish these goals? Why might this scenario be more dangerous than the first three?

   Note: You can restart the server and clear the list of messages that have already been submitted by pressing `CTRL+C` and then either re-running `run_server.sh` or re-running the two docker commands listed above.

3. Suppose now you are a pentester who has been hired to assess the security of these web applications. If you were explaining to a developer of these applications why these XSS bugs are bad (and need to be fixed), think about how you would explain the potential consequences of these bugs. Write down your findings in a paragraph of text.

   For instance, "The application is vulnerable to Y Cross-Site Scripting attack, in the X component/feature. This vulnerability could allow an attacker to Z by . . . etc.".

3 Understanding the Web App Implementation

The web application has the following (very simple) structure:

app.py A Python file that implements the logic of the web application, in the Flask web application framework. It defines the "routes" that the application serves (which can be thought of as the individual "pages" or pieces of application functionality) and what the application does in response to GET/PUT/etc. requests to each.

templates/*.html These are HTML templates: HTML files with placeholders for code and content that will execute or be populated at runtime.

This application contains two templates: `login.html` is the login page, while `msg.html` is the main application page.

Templates can include (at least) two kinds of code: server-side and client-side. For the demos we are looking at, the server-side code is written in Jinja2. For instance, the following code is executed by the server when the user has set a filter keyword to tell the user that messages are being filtered by that keyword.

```
{% if request.args.get('filter') %}
  <p>Displaying only messages that contain <b>{{ request.args.get('filter')|safe
 }}</b> ...
```

On the other hand, the following JavaScript code in the login form is executed on the client to display the message to the user when the password is incorrect:

```
    function displayMsg() {
        const params = new URLSearchParams(window.location.search);
        if (params.has("msg")) {
            msg = params.get("msg");
            document.getElementById("msg").innerHTML = msg;
        }
    }
```

4 Mitigations

Mitigations can be applied either on the client or the server. Certain kinds of mitigations work only for certain kinds of XSS vulnerabilities.

4.1 Auto-Escaping Templates

Server XSS (e.g. Server Reflected or Server Stored XSS) can be mitigated by ensuring that the server does not blindly render untrusted content into the page wihtout first attempting to ensure the content is safe. One way to help do this is by *escaping* the content, e.g. by replacing "<" with "<" etc.

The default templating engine (which performs server-side processing of content in our web applications) performs this kind of escaping by default. In fact, we had to turn it off to make the Server XSS vulnerabilities work.

To see how auto-escaping templates can help mitigate XSS, edit the `msg.html` template. Search for the string "|safe", which appears in multiple places in the file, e.g., in:

```
<p>Displaying only messages that contain <b>{{ request.args.get('filter')|safe }}</b>
```

This is a special filter that causes the input in question (in this case the value of `request.args.get('filter')` (the value of the "filter" URL parameter) to be treated as if it were safe. Doing so causes the templating engine to render this content without escaping it.

We can turn auto-escaping back on of course by removing this filter, i.e.:

```
<p>Displaying only messages that contain <b>{{ request.args.get('filter') }}</b>
```

This is what one would write by default. Make this change and try the XSS using the "filter" URL parameter again. If you view the page source you will see that the symbols have been escaped, i.e. < has become &lt.

Does the change mitigate the Server XSS vulnerability?

Would you expect this same kind of mitigation to help against Client XSS? Why?

4.2 Server Input Sanitisation

Another mitigation for Server XSS is to use a sanitisation library to ensure that the server properly sanitises untrusted input before processing it. This should always be done in addition to using auto-escaping templates.

The library we will use is called bleach and is written and maintained by Mozilla, the makers of Firefox. You can find more information about bleach from `https://pypi.org/project/bleach/`

To try this out, edit the `app.py` file. Find the following line, which is responsible for processing the stored messages that users have posted.

```
        msg = tag + request.form['msg']
```

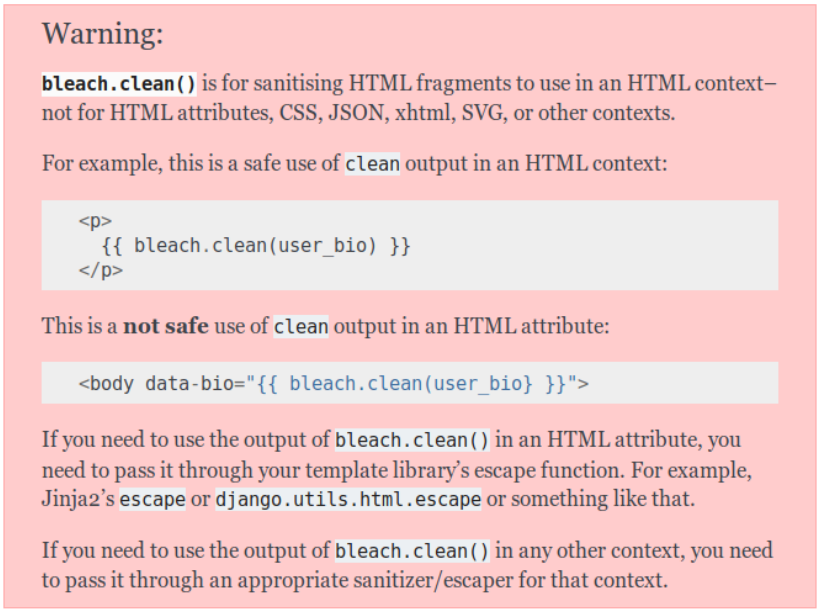
Replace it with the following line, which now uses the bleach library to first sanitise the message contents before storing it.

```
      msg = tag + bleach.clean(request.form['msg'])
```

Then try the broadcast message application. Does this mitigation prevent the XSS due to stored user messages, even without using auto-escaping templates?

In a large web application you will need to sanitize all inputs to be safe. Some things to remember:

- Switch on template escaping (most template engines have it on by default)
- However, don't rely on the template engine. In particular, don't store potentially dangerous input into your database/files. Not only is that dangerous for other reasons (as we will be looking at in the Week 2 lectures), but you also might access that data outside of a template. For example, you might add functionality to get data via an AJAX query, which won't pass through a template. Or you might directly access a value in a manually constructed page - this is common in error pages which often aren't styled, so don't go through templating.
- Know the limits of your sanitization library, to do this you need to read the documentation `https://bleach.readthedocs.io/en/latest/clean.html`. In the case of bleach, be aware it is for HTML tags only, note this warning on the front-page of the documentation:



Warning:

`bleach.clean()` is for sanitising HTML fragments to use in an HTML context– not for HTML attributes, CSS, JSON, xhtml, SVG, or other contexts.

For example, this is a safe use of `clean` output in an HTML context:

```
<p>
  {{ bleach.clean(user_bio) }}
</p>
```

This is a **not safe** use of `clean` output in an HTML attribute:

```
<body data-bio="{{ bleach.clean(user_bio} }}">
```

If you need to use the output of `bleach.clean()` in an HTML attribute, you need to pass it through your template library's escape function. For example, Jinja2's `escape` or `django.utils.html.escape` or something like that.

If you need to use the output of `bleach.clean()` in any other context, you need to pass it through an appropriate sanitizer/escaper for that context.

4.3 Client Sanitisation

For Client XSS, neither auto-escaping templates nor server-side sanitisation are sufficient. This is because in Client XSS the untrusted data is not being processed on the server. Instead, we need to apply mitigations on the client (i.e. to the JavaScript code that is processing the untrusted input).

One way to do this is to carefully handle untrusted input and to ensure that all DOM manipulation performed by client-side JavaScript does not allow unwanted content to be injected into the DOM. You can see an example of this in the `login.html` template. In particular, following lines (which are commented out) show how to safely handle the "msg" parameter:

```
var node = document.createTextNode(msg);
document.getElementById("msg").appendChild(node);
```

Here we explicitly create a text node to hold the untrusted content and add it to the DOM.

Another approach is to use client side input sanitisation. This can be especially useful when processing content that must be URI decoded.

For instance, in the `msg.html` template, the URI hash component stores the colour scheme. It is URI decoded before it is processed. This is done by the following code:

```
var style = decodeURIComponent(window.location.hash.substring(1))
```

We can attempt to mitigate the XSS due to the colour scheme by sanitising the URL hash component using using the popular JSXSS library (`https://jsxss.com/en/index.html`. This library has been included in the `static` folder.

You can enable it by calling the `filterXSS()` function to filter the URI decoded hash component, by adding the following line after the one above:

```
style = filterXSS(style)
```

Does this fully prevent the risks of injection using the URL hash component?

If you investigate carefully, you should find that using sanitisation here alone doesn't fully protect against the risk of injection.

Much like the bleach library you need to be careful about context. If you are using untrusted input for attributes or CSS (as we are here) you will need to escape them in the appropriate context. For further examples, see Rule 2 through 4 at `https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.md`

In particular note the advice when setting attribute values from untrusted sources "Except for alphanumeric characters, escape all characters with ASCII values less than 256". One easy way to do that is to URI encode them e.g. using the `encodeURIComponent()` JavaScript function.

Therefore to fully protect against the XSS due to the colour scheme, we need to replace the new line above with:

```
style = encodeURIComponent(filterXSS(style))
```

4.4 Content Security Policy (CSP)

CSP is a mitigation that can be applied to mitigate all of the XSS vulnerabilities, because this mitigation can prevent injected JavaScript from executing.

In the `login.html` and `msg.html` templates you will find the following line commented-out:

```
        <meta http-equiv="Content-Security-Policy" content="default-src 'self'";>
```

This line applies a pretty restrictive security policy to prevent execution of scripts and content that come from sources outside of the application itself. This may be appropriate for simple applications, but more complex applications need more complex security policies and getting the policy right can be challenging.

Uncomment this line and see what impact it has in each of the four XSSs (after removing the earlier mitigations). Does it prevent injection of JavaScript? Does it impact other functionality of the application?

You should open the Developer/JavaScript Console of your web browser to look for error messages that indicate what is happening when this policy is being applied. Try injecting JavaScript in each of the applications and see what happens.

5 Summary

By the end of the workshop you should have seen a number of examples of XSS, with some knowledge of how dangerous they can be (by having thought about how they can be exploited), as well as experienced the deployment of some of the mitigation strategies. This is by no means an exhaustive list of possible XSS, further examples and preventative measures can be found in the OWASP documentation and cheat sheets.

5.1 Additional Activities

If you finish working through all the examples you could look at Google's XSS game: `https://xss-game.appspot.com/`.

You can also think about more complicated sanitisation scenarios, including:

- Think about how to sanitize a stylesheet entry, refer to the OWASP cheat sheet to think of methods on the client or server to protect such values - is it even possible to fully protect against XSS in such contexts?
- Create a more advanced CSS sanitizer - what if you want to provide a CSS class attribute that includes a space? Is it possible to do that safely?