Cross-Site Scripting (XSS)

Cross-Site Scripting stems from a lack of encoding when information gets sent to application's users. This can be used to inject arbitrary HTML and JavaScript; the result being that this payload runs in the web browser of legitimate users. As opposed to other attacks, XSS vulnerabilities target an application's users, instead of directly targeting the server.
Some examples of exploitation include:

- injecting a fake login form;
- retrieving legitimate users' cookies;
- injecting browser's exploits;
- getting users to perform an arbitrary action in the web application;

In this section, we will only focus on the detection of Cross-Site Scripting. You will have to wait for a full exercise on this subject to get more details on how to exploit these vulnerabilities.
The easiest, and most common proof that a XSS vulnerability exists is to get an alert box to pop up. This payload has many advantages:

- it shows that JavaScript can be triggered;
- it's simple;
- it's harmless.
-

To trigger a pop-up, you can simply use the following payload:

```
alert(1).
```

If you are injecting inside HTML code, you will need to tell the browser that this is JavaScript code. You can use the `<script>` tag to do that:

```
<script>alert(1);</script>.
```

When testing for XSS, there are two important things to remember:

- The response you get back from the server is probably not the only place this information will be echoed back. If you inject a payload and you get it back correctly encoded in page A, it doesn't mean that this information will be correctly encoded in page B.

- If you find a problem with encoding, but can't get your XSS payload to run, someone else may be able to. It's always important to report an encoding problem, even if some protection prevents you from getting your payload from executing. Security is an evolving domain, with new tricks published every week. Even if you cannot exploit a XSS vulnerability now, you or someone else may be able to get another payload to work later on.

There are three types of XSS:

- Reflected: the payload is directly echoed back in the response.
- Stored: the payload can be echoed back directly in the response but will more importantly be echoed back in the response when you come back to this page or to another page. The payload is stored in the backend of the application.
- DOM-based: the payload is not echoed back in the page. It gets executed dynamically when the browser renders the page.

When testing for XSS, you need to read the source of the HTML page sent back, you cannot just wait for the alert box to pop up. Check what characters get encoded and what characters don't get encoded. From this, you may find a payload that works.

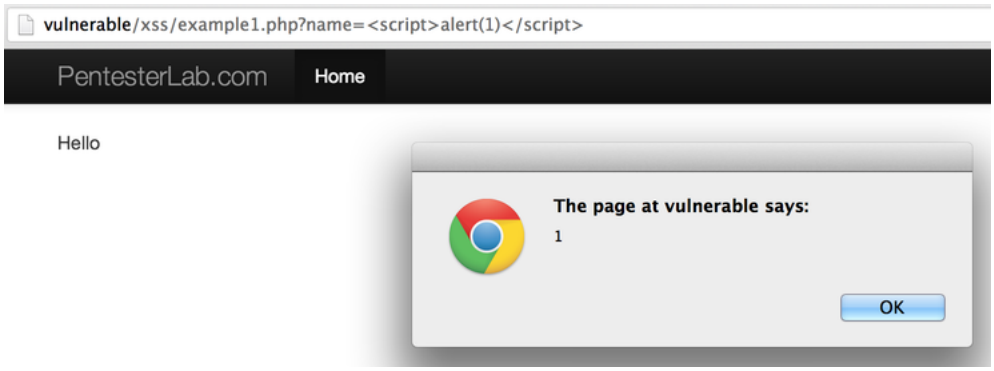
Some browsers provide built-in protection against XSS. This protection can be enabled or disabled by the server (it has been disabled in the ISO). If you find that your payload is directly echoed back in the page but no alert box pops up, it's probably because of this protection. You can also disable this protection by telling your browser to disable it. For example, in Chrome, it can be done by running Chrome with the option

`--disable-xss-auditor`.

Task 1 - Example 1

The first vulnerable example is just here to get you started with what is going on when you find a XSS. Using the basic payload, you should be able to get an alert box.

Once you send your payload, you should get something like:



Make sure that you check the source code of the HTML page to see that the information you sent as part of the request is echoed back without any HTML encoding.

Task 2 - Example 2

In the second example, a bit of filtering is involved. The web developer added some regular expressions, to prevent the simple XSS payload from working.

If you play around, you can see that <script> and </script> are filtered. One of the most basic ways to bypass these types of filters is to play with the case: if you try <sCript> and </sCRIpt for example, you should be able to get the alert box.

Task 3 - Example 3

You notified the developer about your bypass. He has added more filtering, wich now seems to prevent your previous payload. However, he is making a terrible mistake in his code (which was also present in the previous code)...

If you keep playing around, you will realise that if you use Pentest<script>erLab for payload, you can see PentesterLab in the page. You can probably use that to get <script> in the page, and your alert box to pop up.

Task 4 - Example 4

In this example, the developer decided to completely blacklist the word

script: if the request matches script, the execution stops.

Fortunately (or unfortunately depending on what side you are on), there are a lot of ways to get JavaScript to be run (non-exhaustive list):

- with the <a tag and for the following events: onmouseover (you will need to pass your mouse over the link), onmouseout, onmousemove, onclick ...
- with the <a tag directly in the URL: <a href='javascript:alert(1)'... (you will need to click the link to trigger the JavaScript code and remember that this won't work since you cannot use script in this example).
- with the <img tag directly with the event onerror: .
- with the <div tag and for the following events: onmouseover (you will need to pass your mouse over the link), onmouseout, onmousemove, onclick...
- ...

You can use any of these techniques to get the alert box to pop-up.

Task 5 - Example 5

In this example, the <script> tag is accepted and gets echoed back. But as soon as you try to inject a call to alert, the PHP script stops its execution. The problem seems to come from a filter on the word alert.

Using JavaScript's eval and String.fromCharCode(), you should be able to get an alert box without using the word alert directly. String.fromCharCode() will decode an integer (decimal value) to the corresponding character.
You can write a small tool to transform your payload to this format using your favorite scripting language.

Using this trick and the ascii table, you can easily generate the string: alert(1) and call eval on it.

Another easier bypass is to use the functions prompt or confirm in Javascript. They are less-known, but will give you the same result.

Task 6 - Example 6

Here, the source code of the HTML page is a bit different. If you read it, you will see that the value you are sending is echoed back inside JavaScript code. To get your alert box, you will not need to inject a script tag, you will just need to correctly complete the pre-existing JavaScript code and add your own payload, then you will need to get rid of the code after your injection point by commenting it out (using //) or by adding some dummy code (var $dummy = ") to close it correctly.

Task 7 - Example 7

This example is similar to the one before. This time, you won't be able to use special characters, since they will be HTML-encoded. As you will see, you don't really need any of these characters.
This issue is common in PHP web applications, because the well-known function used to HTML-encode characters (htmlentities) does not encode single quotes ('), unless you told it to do so, using the ENT_QUOTES flag.

Task 8 - Example 8

Here, the value echoed back in the page is correctly encoded. However, there is still a XSS vulnerability in this page. To build the form, the developer used and trusted PHP_SELF which is the path provided by the user. It's possible to manipulate the path of the application in order to:

- call the current page (however you will get an HTTP 404 page);
- get a XSS payload in the page.

This can be done because the current configuration of the server will call /xss/example8.php when any URL matching /xss/example8.php/... is accessed. You can simply get your payload inside the page by accessing /xss/example8.php/[XSS_PAYLOAD]. Now that you know where to inject your payload, you will need to adapt it to get it to work and get the famous alert box.

Trusting the path provided by users is a common mistake, and it can often be used to trigger XSS, as well as other issues. This is pretty common in pages with forms, and in error pages (404 and 500 pages).