In the DOM We Trust: Exploring the Hidden Dangers of Reading from the DOM on the Web

Jan Drescher*
Technische Universität Braunschweig
Braunschweig, Germany
jan.drescher@tu-braunschweig.de

Sepehr Mirzaei*
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
sepehrmirzaei98@gmail.com

Soheil Khodayari
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
shl.khodayari@gmail.com

David Klein
Technische Universität Braunschweig
Braunschweig, Germany
david.klein@tu-braunschweig.de

Thomas Barber
SAP SE
Karlsruhe, Germany
thomas.barber@sap.com

Martin Johns
Technische Universität Braunschweig
Braunschweig, Germany
m.johns@tu-braunschweig.de

Giancarlo Pellegrino
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
pellegrino@cispa.de

Abstract

The DOM tree is a central part of modern web development, enabling JavaScript to interact with page content and structure. Only a few prior studies have studied its trustworthiness, despite its widespread use in guiding program logic and security decisions. Most notably, script gadgets have shown how this trust can be exploited by triggering the execution of benign JavaScript fragments with seemingly harmless markup injections. In this paper, we show that script gadgets are only the tip of the iceberg. Seemingly-benign markup injections can trigger the execution of fragments—that we call **DOM gadgets**—that, unlike script gadgets, do not necessarily result in a cross-site scripting vulnerability. Instead, they can result in a broader set of attacks, such as browser request hijacking attacks, cross-site request forgery attacks, and user interface manipulations.

In this paper, we introduce an automated approach that combines static and dynamic analysis to detect DOM gadgets, tracing flows from the DOM to security-sensitive sinks, and assessing the presence of validation or sanitization checks. We conduct a large-scale web crawl across the top 15k domains and identify 2.6 million DOM-to-sink data flows that could lead to DOM gadget exploitation. We complement this by automatically detecting markup injection vulnerabilities, finding 657 DOM gadgets on 37 sites with the markup injection vulnerability required to exploit the DOM gadget. We further analyze these flows to assess the presence and effectiveness of security checks, revealing that 10% of DOM gadget flows receive no validation or sanitization checks. Our results indicate that DOM-based input trust is both widespread and underprotected. Our work highlights the scale and diversity of DOM gadget vulnerabilities in the wild, motivating a rethink of the DOM's role in web application trust boundaries and offering tools to aid in their identification and mitigation.

CCS Concepts

• **Security and privacy** → **Web application security**.

Keywords

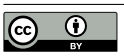
DOM; Script Gadgets; Prevalence; DOM Gadgets

ACM Reference Format:
Jan Drescher, Sepehr Mirzaei, Soheil Khodayari, David Klein, Thomas Barber, Martin Johns, and Giancarlo Pellegrino. 2025. In the DOM We Trust: Exploring the Hidden Dangers of Reading from the DOM on the Web. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765117

1 Introduction

The Document Object Model (DOM) [11] is a programming interface central in web development that is primarily used to dynamically modify the content and appearance of webpages. More recently, the DOM has also been used to store data, including sensitive data items such as site configurations – often embedded in element properties like `data-*` attributes. Developers leverage a variety of DOM query functions, such as `querySelector`, to retrieve these elements and use their attributes to perform security-relevant operations, such as generating URLs to fetch external resources. Unfortunately, attackers can exploit such DOM read operations by injecting seemingly benign HTML elements and hijacking the control flow of benign code snippets to achieve unauthorized cross-site requests or, worse, arbitrary JavaScript execution. To date, the security implications of reading data from the DOM tree remain largely unexplored.

*Both authors contributed equally to this research.



CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765117

The security risks when interacting with the DOM tree have been studied before, with most prior attention dedicated to *write* operations. The most notorious examples of attacks are DOM-based (or client-side) Cross-Site Scripting (XSS) [27, 50, 55, 68], where attacker-controlled input strings, typically originating from the page URL, are inserted in the DOM tree, enabling attackers to achieve arbitrary JavaScript execution. Only recently has the focus shifted towards more subtle code execution attacks originating from unexpected interactions between the DOM tree and developers' code. In these attacks, attackers no longer inject malicious JavaScript code; instead, they use seemingly benign HTML elements that can hijack JavaScript control flow when injected into the DOM tree. One such attack is DOM Clobbering [44], where the attacker leverages a naming collision between JavaScript variables and named HTML markups, resulting in the page's scripts implicitly accessing attacker-controlled data in the DOM. Another example of such attacks is script gadgets [49], which are fragments of benign JavaScript that explicitly access and consume DOM elements and can execute JavaScript when attackers control such elements. Despite all these works, these threats have largely focused on attacks that ultimately result in the execution of malicious JavaScript code, leaving open the question of whether—and to what extent—other attacks beyond code execution are possible.

In this paper, we look at the broader category of vulnerabilities and attacks that originate when fragments of benign JavaScript code explicitly consume DOM elements. We call this category of vulnerabilities **DOM gadgets**. As a first step, we undertake a systematization of possible DOM gadgets beyond the script gadgets, along with the attacks they enable. Then, we perform one of the first large-scale measurements of DOM gadgets in the wild, aiming to determine the extent to which developers trust the DOM via a multi-step hybrid data-flow analysis. First, we identify the DOM gadgets searching for code fragments that (i) read from the DOM tree via DOM queries and (ii) perform sensitive operations. We do so by analyzing the data flows from the DOM tree to sensitive operations. Then, we identify the markup injection automatically by analyzing the data flows from untrusted sources, e.g., the URL, to the DOM tree, which is contrary to prior works (i.e., [44, 49]) that assumed that an injection point exists for the gadgets. Finally, we identify the pages having both DOM gadgets and the required markup injection.

We applied our methodology to the top 15K Tranco websites, collecting a large dataset of 522K webpages and 10.3B lines of JavaScript code. Our results show that DOM gadgets are ubiquitous in the wild, with an overall 357K verified instances affecting ~15% of the analyzed domains, of which 77% correspond to new gadget types beyond script gadgets. By automatically identifying injection points for the DOM gadgets, we identified 657 DOM gadgets with the required markup injection across 37 sites. These gadgets can be used for malicious purposes beyond injecting JavaScript, including hijacking outgoing requests, web sockets, and top-level navigation URLs—among the most frequent ones. We analyzed 60K gadgets found via static analysis, uncovering that 10% of the gadgets perform no input sanitization operation. We present four attack techniques to reorder DOM elements and exploit DOM gadgets, showing that these new techniques are a requirement for exploiting at least 34% of the discovered gadgets.

In summary, this paper makes the following contributions:

- We propose the first characterization and analysis of **DOM gadgets**, the broader category of vulnerabilities and attacks that originate when fragments of benign JavaScript code consume DOM elements.
- We propose one of the first analysis techniques targeting gadget-based vulnerabilities, where we automated both the identification of the gadgets and verified their exploitability by identifying injection opportunities for the attacker.
- We performed a large-scale analysis of the top 15K popular Tranco sites, identifying 357K verified DOM gadgets affecting ~15% of the analyzed sites, from which we found 657 DOM gadgets spanning 37 sites with an injection point.
- We present four novel attack techniques to exploit DOM gadgets, which enable reordering of DOM elements, and show that these new techniques are necessary for at least 34% of the discovered gadgets.

Open Science Statement—Please see §8.4

2 Background

Before presenting our study, we describe the background information required to understand this work.

2.1 Script Gadget Vulnerability

Script gadgets [49] are code fragments within client-side JavaScript programs that unexpectedly react to code-less nodes injected into the DOM, reading properties of injected nodes and using them in code execution sink instructions, such as `eval` [36]. In a sense, script gadgets transform the initially benign markup into executable code, presenting a new and code-less breed of client-side Cross-Site Scripting (XSS) vulnerabilities [31, 45, 50, 68]. XSS attacks are a critical threat to web applications, allowing adversaries to exfiltrate sensitive data, manipulate application behavior, or perform unauthorized actions on behalf of user victims–to name only a few examples.

Traditional XSS vulnerabilities arise when applications fail to properly validate untrusted input containing code, typically solved by *controlling* or *disallowing* code execution, such as input sanitization [34, 45] and Content Security Policy (CSP) [67, 72]. In code-less XSS, the input does not contain malicious JavaScript code directly but can hijack the execution of existing client-side JavaScript code through a script gadget. Unlike the traditional XSS, existing XSS countermeasures are insufficient to protect web applications from these new XSS variants [49]. Modern input sanitizers [29, 34, 45], like DOMPurify [34] and the new sanitizer API [29] can only sanitize inputs containing JavaScript code, which is not the case for script gadgets. On the other hand, the CSP cannot prevent the execution of already-present code that reacts to code-less markups. These observations suggest that existing countermeasures may be incomplete. The research community has only recently started exploring the impact of script gadgets on the security posture of web applications [49, 56].

```
1  <script>
2  var cartItems =
3      document.querySelectorAll('.cart-item');
4  for (const elem of cartItems) {
5      var url = elem.getAttribute('data-url');
6      // check item inventory and price
7      fetch(url, {
8          method: 'POST',
9          headers: { 'XSRF-Token': "xyz" }),
10         body: JSON.stringify({ ... }),
11     }).then(resp => { /* [...] /* });
12 }
13 </script>
14 <div class="cart-item"
15   data-url="/api/v1/checkInventory?id=item-12345">
16 </div>
```

Listing 1: Example of a DOM gadget vulnerability in a shopping cart application targeting the "Add to Cart" functionality.

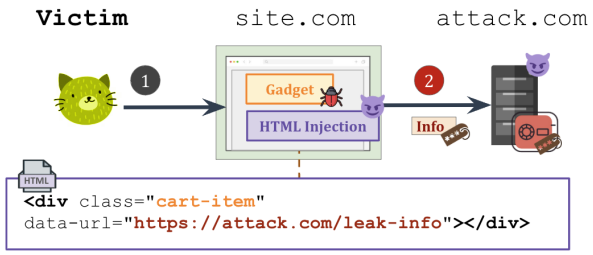
2.2 DOM Gadget Vulnerability

Script gadgets are just one instance of a broader, largely unexplored issue involving vulnerable gadgets present within client-side JavaScript programs. In this case, a property of an injected node flows to a code execution instruction, i.e., XSS. While significant attention has been given to code execution risks, the threats stemming from other sensitive APIs and operations—such as asynchronous requests, web sockets, event sources, post messages, and top-level navigations—remain unexplored in the context of DOM gadget exploitation. For example, attackers can abuse these gadgets to obtain client-side request forgery [1, 42], cross-site socket hijacking [39, 60], and information leakage [32, 39, 69].

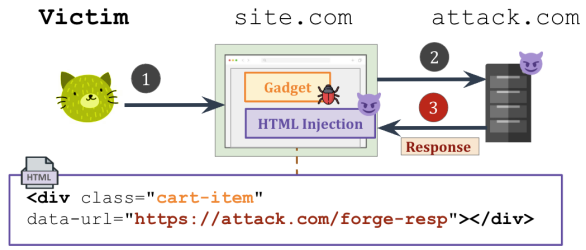
2.2.1 Vulnerability Description. A DOM gadget vulnerability occurs when a JavaScript program selects a node from the DOM tree using a specific DOM selector, retrieves the value of a property from the selected node, and then uses this value in a security-sensitive operation without proper validation, enabling attackers to execute arbitrary code or perform unintended operations. The fundamental issue lies in developers mistakenly assuming that the content within the DOM is inherently trustworthy. This leads them to use it directly in sensitive operations without proper validation, creating opportunities for attackers to manipulate the DOM and exploit these gadgets.

Listing 1 shows an example of a DOM gadget vulnerability in a shopping cart application. The client-side code dynamically updates the cart whenever an item is added. When a new item is added to the DOM (e.g., after selecting a product), the application triggers a request to check the inventory. First, it selects all nodes with the `cart-item` class (lines 2-3). When such a node is detected, it retrieves the `data-url` attribute (line 5) and sends a POST request to the specified URL (lines 7-11) to check the inventory and price of the item, identified by the item ID in the URL. The request includes a token (line 9) to authenticate the request against CSRF [26, 43, 51]. The vulnerability stems from the implicit trust developers place in DOM nodes (`cart-item` elements) and using them to generate an authenticated request.

2.2.2 Attack Overview. Attackers can exploit DOM gadgets by injecting one or more seemingly harmless nodes into the DOM that match the selector query used by the JavaScript code. These malicious nodes can manipulate the program's behavior by being selected in place of legitimate nodes and, consequently, be used in sensitive instructions. Figure 1 shows example attack scenarios exploiting the vulnerability in Listing 1. If attackers inject malicious nodes with crafted `data-url` attributes, they can control the destination of the asynchronous request (line 7). This allows them to hijack the request and redirect it to their own servers. This has various security implications.



(a) Information Leakage.



(b) Response Forgery.

Figure 1: Example attack exploiting a DOM gadget vulnerability.

Firstly, attackers can exfiltrate sensitive information (Figure 1a), such as the XSRF token embedded in the request header or personally identifiable information (PII) included in the request body, like a user's address used for inventory checks and delivery. With access to the XSRF token, attackers can execute CSRF attacks, forging requests to arbitrary state-changing application endpoints. Secondly, by manipulating the response to these requests, attackers can inject arbitrary data into the application (Figure 1b). For example, they could falsify the response to set a product price to zero, effectively allowing them to acquire goods or services without payment.

2.3 Threat Model

In this paper, we consider a *web* attacker [25, 26] who abuses inputs such as URL parameters, window name, document referrer, and postMessages, to inject code-less HTML markups to the DOM tree, and exploit DOM gadget vulnerabilities present within the page to trigger sensitive instructions, which is in line with prior work in the area of client-side vulnerabilities [42, 43, 49, 50, 69]. To achieve this, the attacker exploits a *Markup injection* vulnerability in the web application that reflects attacker-controlled inputs (e.g., from

URL parameters) into the document. If mitigations like CSP or input sanitization are in effect, the attacker cannot exploit this vulnerability to gain code execution immediately. Instead, they inject markups that are selected and read by the DOM gadget. The attacker in Figure 1 combines the gadget and markup injection vulnerabilities, using the markup injection vulnerability to inject the displayed markup that subsequently triggers the DOM gadget.

In contrast to the strong assumptions made in prior research on script gadgets (i.e., [50]), we do not assume that an adversary can inject code-less HTML nodes into the DOM on all webpages containing such gadgets. Instead, we propose an end-to-end approach to identify specific pages where both markup injection vulnerabilities and DOM gadgets coexist. This approach is more realistic, avoiding broad assumptions about attacker capabilities.

In addition, prior work [50] assumes that attackers must inject their malicious node before the benign one in the DOM tree, as query selectors typically select the first matching element. However, this paper introduces a novel mechanism that eliminates this requirement, expanding the scope of potential exploitation.

3 Problem Statement

Modern web applications increasingly rely on dynamic interactions between client-side code and the DOM, creating opportunities for attackers to exploit vulnerable patterns. While prior work has highlighted the risk of script gadgets enabling code execution [49, 56], this represents only a subset of potential threats posed by DOM gadgets. Browsers support a wide range of sensitive instructions, including asynchronous requests, web sockets, and post messages–to name only a few examples. If attackers can manipulate these instructions, the consequences could range from data exfiltration to unauthorized actions across application states. This raises key research questions about the systematization, detection, and exploitation of these vulnerabilities:

RQ1: Gadget Systematization. Beyond script gadgets, what other types of DOM gadgets exist, and how can attackers exploit them?

RQ2: Gadget Detection and Prevalence. How can we identify DOM gadgets at scale using static and dynamic analysis? How prevalent are these gadgets in real-world applications, and to what extent do developers trust the DOM?

RQ3: Exploitable Gadgets and Impact. How many pages with DOM gadgets are truly exploitable, allowing attackers to inject markup into the DOM to trigger them?

4 Systematization of DOM Gadgets

We now address RQ1, outlined in §3, with the goal of systematizing DOM gadgets.

4.1 Reading from DOM

Client-side JavaScript can access data from the DOM through a variety of APIs, including `document.querySelector` and `document.querySelectorAll`, as specified by the W3C [23] and WHATWG [24] specifications. These APIs allow for node selection in the DOM tree using patterns that describe the desired attributes of target nodes, such as matching id and class names, commonly known as *DOM selectors* [21]. Browsers also offer simpler APIs for accessing DOM content, such as `document.getElementById` and `document.getElementsByClassName`, which are typically streamlined wrappers around DOM selectors. Table 1 summarizes the syntax of DOM selectors, and Table 2 lists the APIs that can use those selectors to read content from DOM.

Table 1: DOM Selectors.

Category	Selector	Matches
Basic Selectors	`E`	element of tag `E`
	`E#foo`	`E` with id `foo`
	`E.foo`	`E` with class `foo`
	`E[foo]`	`E` with an attribute `foo`
Attribute Selectors	`E[foo="bar"]`	`E` with attribute `foo` and value `bar`
	`E[foo~="bar"]`	`E` whose `foo` attribute value contains `bar`
	`E[foo^="bar"]`	`E` whose `foo` attribute starts with `bar`
Function Selectors	`E:first-child`	`E` element that is the first child of its parent
	`E:last-child`	`E` element that is the last child of its parent
	`E:nth-child(n)`	The `n`-th child `E` element of its parent
	`E:not(s1,s2)`	`E` that does not match selectors `s1` or `s2`
	`E:is(s1,s2)`	`E` that matches `s1` or `s2`
	`E:where(s1,s2)`	`E` that matches `s1` or `s2` with no specificity
	`E:has(s1,s2)`	`E` containing an element matching `s1` or `s2`
Relation Selectors	`E > E'`	`E'` that is a child of an `E`
	`E + E'`	`E'` immediately following an `E`

Table 2: APIs for reading from the Document interface via DOM selectors. The last column shows whether our Foxhound implementation supports it.

JavaScript API	Ref.	Foxhound
`document.getElementById(id)`	[22] § 4.2.4	✓
`document.getElementsByName(name)`	[22] § 4.5	✓
`document.getElementsByClassName(className)`	[22] § 4.5	✓
`document.getElementsByTagName(tagName)`	[22] § 4.5	✓
`document.getElementsByTagNameNS(tagName)`	[22] § 4.5	✓
`document.querySelector(selector)`	[22] § 4.2.6	✓
`document.querySelectorAll(selector)`	[22] § 4.2.6	✓
`document.elementFromPoint(x, y)`	[22] § 4.5	✓
`document.elementsFromPoint(x, y)`	[22] § 4.5	✓

Another common method to read content from the DOM is through event handlers, specifically by accessing objects on which events are fired via the EventTarget interface [15]. This approach is analogous to the example presented in Listing 1. Finally, there exist other methods for reading content from the DOM, such as XPath expressions [10] and node navigation through parent/child relationships. However, these approaches are too brittle and prone to break with minor changes in the UI, making it significantly harder for attackers to exploit them. In this work, we focus on query selector-based DOM read APIs.

4.2 DOM Gadgets and Vulnerabilities

DOM gadgets are vulnerable data flow patterns in client-side JavaScript, where attacker-controlled inputs from DOM nodes ultimately reach various sensitive APIs (sinks), resulting in a wide range of security issues. Script gadgets are one instance of DOM gadgets, where the affected API is a JavaScript code execution instruction, leading to XSS. While script execution is the most immediate concern, the definition of script gadgets overlooks a wide

Table 3: Overview of DOM gadgets and attacks. Legend: ⊕ = new DOM gadget variants.

	Gadget	XSS	Content Manip.	Phishing	Unauth Action	Info Leak	Session Hijack.	Open Redirect	Drive-by Downl.	Rogue Plugin	Related Ref.
	Code Execution	●	○	○	○	○	○	○	○	○	[49, 50]
	Markup Injection	●	●	●	○	○	○	○	○	○	[34, 45, 49]
⊕	Async. Request	○	○	○	●	●	○	○	○	○	[1, 39, 42]
⊕	WebSocket	○	○	○	●	●	●	○	○	○	[9, 54, 60]
⊕	Navigation	●	○	●	●	○	●	●	○	○	[40]
⊕	Object Loading	●	○	○	○	●	○	○	●	●	[47, 66, 71]
⊕	Form/Link Manip.	●	○	●	○	●	○	●	○	●	[4, 35]

array of other vulnerabilities. Characterized by different sinks, exploit techniques and impact, these overlooked gadgets from the majority of all DOM gadgets.

DOM gadgets can enable attackers to manipulate DOM-based sinks for purposes such as request hijacking, credential theft, or unauthorized state changes. We reviewed W3C [23] and WHATWG [24] specifications, as well as academic and non-academic literature (see, i.e., [1, 4, 9, 34, 35, 39, 40, 42, 45, 47, 49, 50, 54, 60, 66, 71]), looking for Web APIs and instructions that can be manipulated by DOM gadgets. We categorized the potential threats of DOM gadgets based on the sensitive instructions they exploit. Table 3 summarizes our findings. Below, we describe each gadget type and its threats.

Code Execution Gadgets. These gadgets enable attackers to execute arbitrary code by leveraging instructions that evaluate or execute strings as code, such as `eval`, `new Function()`, and `setTimeout` [49, 50]. They are commonly used in XSS attacks to inject and run malicious JavaScript, compromising user data and application integrity. Code Execution gadgets are a form of script gadgets [49].

Markup Injection Gadgets. These gadgets manipulate the structure or content of the DOM, potentially injecting malicious content or altering page behavior [34, 49]. These gadgets exploit dynamic markup insertion instructions such as `innerHTML`, `document.write`, and `iframe.srcdoc`, allow attackers to inject malicious HTML. This can lead to client-side XSS, unauthorized UI changes, and phishing attacks through content manipulation. Markup injection gadgets are another form of script gadgets [49].

Asynchronous Request Gadgets. Asynchronous request gadgets exploit APIs like `fetch`, `XMLHttpRequest`, and `navigator.sendBeacon`, which facilitate communication with web services such as REST APIs without reloading the page. If attackers gain control over the URL, body, or headers of these requests, they can force victims to perform unauthorized actions, resulting in client-side CSRF attacks [42].

Beyond unauthorized actions, manipulating asynchronous request URLs can also cause sensitive information leakage. By redirecting requests to attacker-controlled servers, attackers can capture sensitive data included in headers or request bodies, such as personally identifiable information (PIIs), or CSRF tokens [39].

WebSocket Gadgets. WebSocket gadgets exploit the WebSocket API, which establishes full-duplex, event-driven communication between browsers and servers, which is exempt from the Same-Origin Policy, typically initiated via an HTTP GET request. If an attacker gains control over the WebSocket connection, they can execute Cross-Site WebSocket Hijacking (CSWSH) [9, 54, 60]. In such attacks, an attacker embeds a WebSocket connection to a target website within a malicious page. When a victim visits the page, their browser performs authenticated actions on the attacker's behalf. Unlike traditional CSRF, CSWSH enables both read and write access to the victim's session. Moreover, if an attacker can manipulate the URL used for the initial WebSocket handshake, they can redirect the connection to a malicious server, enabling information leakage and unauthorized data exchange. Additionally, control over the data sent through the WebSocket allows message hijacking, potentially triggering CSRF-like behaviors. This highlights how WebSocket gadgets pose risks beyond other network requests, amplifying both data theft and abuse scenarios.

Top-level Navigation Gadgets. Top-level navigation gadgets exploit APIs such as `location` and `window.open` to manipulate browser navigation and trigger HTTP requests. The location API can alter the current URL and initiate a new HTTP GET request. If an attacker gains control over the entire URL, they could exploit the `javascript:` protocol for client-side XSS attacks [40] or redirect the browser to a malicious site [41], facilitating phishing or session hijacking. Even partial control of the URL, such as modifying query parameters, can lead to CSRF when state-changing GET requests are supported or when POST requests are improperly accepted as GET. Similarly, the `window.open` API initiates top-level HTTP requests in new or existing browser contexts, posing risks like open redirects, CSRF, and client-side XSS.

Object Loading Gadgets. These gadgets exploit elements responsible for loading external resources such as media, scripts, or objects. This may result in drive-by downloads, rogue plugin injections, inclusion of harmful media, and XSS if attackers can manipulate the URL of dynamically loaded scripts, e.g., via the `script.src` API.

Form/Link Manipulation Gadgets. These gadgets allow attackers to modify form destinations and links [4, 35] through APIs like `form.action` and `a.href`. This can exfiltrate user-entered data, redirect users to malicious pages, and create deceptive phishing links for social engineering. Furthermore, attackers could exploit the `javascript:` scheme for client-side XSS attacks [40].

In summary, our gadget systematization demonstrates the broad spectrum of potential DOM gadgets beyond traditional script gadgets. Attackers exploiting such vulnerabilities can go beyond code injection to cause significant harm, including unauthorized requests, phishing, and data leakage.

4.3 Gadget Exploitation

To exploit the gadgets enumerated in §4.2, attackers often need to hijack the result of DOM query selectors (instructions that read from the DOM tree), by injecting a crafted markup into the webpage, such as the one discussed in §2.2.2 for Listing 1. However, DOM APIs like `querySelector()` typically operate by selecting the first element in the DOM tree that matches the given query. This behavior imposes a significant limitation on attackers for non-event-based DOM reads: they must inject their malicious node before the benign one in the DOM tree to ensure it is selected. However, achieving this precise ordering is not always feasible in real-world scenarios. In this paper, we propose **novel attack strategies** that reduce or even eliminate the above requirement.

Attack 1: Body Element. This technique takes advantage of how browsers handle the insertion of `body` elements into the DOM. By injecting a `body` element with specific properties matching the query (such as id and class name), the browser automatically copies the attributes of the new `body` element to the existing `body` element, even if it was originally injected at the end, which is consistent with the HTML specifications [16, 22]. This allows the node to be reliably selected via DOM APIs like `querySelector()`.

Attack 2: HTML Element. Inside the `body` of an element, when the parser encounters an opening `html` tag, it copies all its attributes to the outer `html` element. This ensures that a querySelector on e.g., an id, always matches the copied attribute.

Attack 3: Table Element. Another way to move an injected node before existing nodes is inside a `table` context. The `table` element constrains the valid child elements, and everything else is moved in front of the table. So if one has injection capabilities in the third row and the target element is in the first, it suffices to inject a `div` tag, which can not occur as a direct child of the table element. Consequently, the HTML parser will move it in front of the currently open `table` due to what is called "foster parenting" [5]. This allows the attacker to overcome some order restrictions, as the injected tag can move in front of tags that regularly occur prior to it.

Attack 4: Frameset Element. In case the `body` element was implicitly created, it is possible to remove all regular content and replace it with a frameset element. This allows to "delete" prior DOM nodes in some specific cases. However, it greatly restricts the DOM structure that can be inserted by the attacker as part of the attack.

Together, these techniques significantly broaden the attacker's capabilities, bypassing the need for precise injection ordering. We quantify the contribution of these new techniques for exploitability in §6.6.

These techniques are fundamentally similar to concepts used for modern mXSS attacks, such as those described by Klein and Johns [46]. However, they serve a completely different purpose. When abusing DOM parsing particularities for mXSS, the goal is to confuse a sanitizer and bypass it by "hiding" the payload. In our case, the goal is completely different. These tricks aim to ensure that a DOM selector selects the injected attribute even in cases where the injection context would normally prevent this.

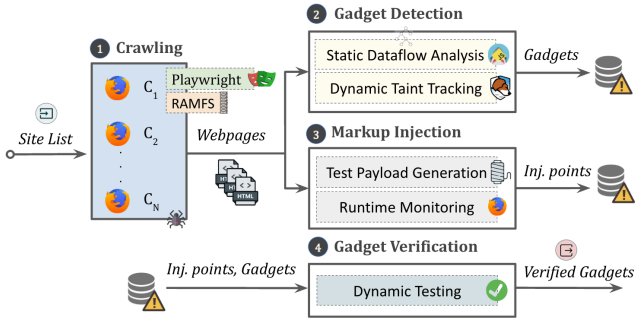


Figure 2: Overview of our methodology.

5 Vulnerability Detection

This section presents our approach to detecting DOM gadgets at scale. Figure 2 presents an overview of our methodology, which comprises three steps: (1) Web Crawling, where a Firefox-based crawler gathers snapshots of webpages (2) Gadget Detection, combining dynamic analysis with a taint-aware browser and static analysis via Code Property Graphs to track data flows and identify DOM gadgets; and (3) Exploitation and Markup Injection, where we test whether markup injection is possible in webpages containing the identified gadgets to exploit them. The rest of this section describes each step in more detail.

5.1 Web Crawling

Starting from a list of seed domains, we created a Playwright-based [18] crawler to collect snapshots of webpages, including JavaScript code and runtime information, such as DOM snapshots and HTTP response headers. The crawler collects webpages of different domains following a round-robin strategy, reducing the load on resource servers (see [64]).

For each domain, it extracts a list of pages and visits them following a depth-first order without URL encoding. It continues until no further pages are in the queue, a maximum of n =100 pages have been visited, or the time budget of t = 30 minutes has elapsed, whichever condition occurs first. For scalability reasons, our crawling infrastructure uses w = 100 workers in parallel while minimizing disk I/O by leveraging RAMFS for concurrent writes.

5.2 Gadget Detection

We formulated the problem of detecting DOM gadgets as a data flow analysis problem, where we intend to track the propagation of attacker-controlled values from JavaScript DOM read operations to sensitive instructions. To balance accuracy and coverage, we used both dynamic and static analysis to detect such data flow patterns. Dynamic analysis enables us to observe actual runtime behavior and detect flows manifesting during execution. In contrast, static analysis provides a broader coverage by examining potential flows based on code structure. We merged the results from both approaches, deduplicating flows by comparing key attributes such as source and sink locations, the types of sources and sinks involved, and the associated URL of the analyzed webpage.

5.2.1 Dynamic Data Flows. To measure dynamic data flows, we created an extended version of the taint-aware Foxhound browser [19, 45]. Specifically, we enhanced Foxhound to consider all read operations from the DOM tree as sources, as indicated in Table 2. This effectively allows us to taint elements in order to be able to detect DOM gadgets during the crawling process. An important benefit of dynamic analysis is that it has little to no false positives, i.e., every data flow recorded by Foxhound actually took place on the page. Additionally, because Foxhound has full control over the JavaScript runtime, information that might be obfuscated in the source code is available. A simple example is that we can record the arguments for query selector calls with Foxhound.

5.2.2 Static Data Flows. We relied on the static analysis engine of JAW [39, 42] to detect DOM gadgets. JAW creates a canonical, graph-based model of the JavaScript program, known as Code Property Graph (CPG) [73], which represents the program syntax and semantics (control and data flows). We extended JAW by creating queries to traverse the CPG searching for DOM gadget data flow patterns. We ran static analysis on 10 unique pages of each website to balance coverage and the high analysis time required by JAW, which is consistent with prior work [41]. Static analysis helps us achieve improved code coverage, especially when DOM gadgets are not executed on page load or are gated by conditions or user interactions.

5.3 Markup Injection

After identifying pages with DOM gadgets, we need to find an injection point to exploit them. To this end, we investigate which pages in our dataset are susceptible to markup injection vulnerabilities. Our taint-aware crawler also collects data flows from web attacker sources, like URL and window name, to markup injection instructions. To identify exploit payloads, we used a similar approach as prior work [27], where we construct a payload by first breaking out of any context that might inhibit the injection of an HTML tag. This break-out sequence is constructed based on the information available through Foxhound, as we both know the tainted part of the string and any surrounding text. Afterwards, we either insert an XSS payload based on the methodology presented in [27] or a `div` tag with a custom `data-` attribute to test for markup injection in cases where an XSS payload is blocked, e.g., by a sanitizer [34, 45].

Afterward, we test the markup injection payload via runtime monitoring and dynamically confirm that it works in a browser. Finally, this component outputs markup injection vulnerabilities that do not lead to XSS, particularly those that are mitigated by CSP [72]. We show that these mitigated injection points can become exploitable again using DOM gadgets.

5.4 Gadget Verification

To verify the existence of data flows from the DOM to gadgets identified by Foxhound, we automatically construct benign markup payloads, inject them into the HTML content of the target page before it is parsed and loaded, and check whether each payload has entirely reached its intended gadgets after the page has loaded. The markup is generated based on the selector API type, its query parameter, and the attributes of the originally targeted element. We assign a benign payload string with a unique identifier to all attributes of the generated markup. This unique identifier is derived from the query expression used in the selector API (source), allowing us to track only the source-sink pairs reported by Foxhound. Additionally, we wrap all functions of the prototypes of all sinks to monitor the arguments passed to these sinks and report any occurrences of our crafted payloads. We only consider a flow as verified when the entire payload reaches the sink—partial matches are excluded. The injection point of the markup payload is determined based on the location of the target element. Once determined, the generated markup is injected accordingly.

For verifying data flows detected by static analysis, we cannot use the same approach as with dynamic flows since static analysis may identify flows that are not immediately triggered on page load and may require user interaction or specific conditions to activate. Therefore, we manually verify the existence of such flows by randomly sampling a subset of flows from each gadget type and inspecting them individually.

5.5 Exploitations and Attacks

After identifying two distinct sets of webpages—one affected by markup injection vulnerabilities and the other containing DOM gadgets—we systematically cross-reference these sets to determine instances where a DOM gadget can be exploited via a markup injection point. This step is crucial in assessing the real-world impact of DOM gadgets. However, not every end-to-end gadget flow can be exploited for an attack. For instance, one of the observed gadgets reads the language for the content to be served from a DOM attribute, sanitizes it using `encodeURIComponent`, and appends it to the URL query parameters of a request. While an attacker can control parts of the URL, namely the `lang` query parameter, this control cannot be repurposed for an attack.

To assess if a gadget can be used for an attack, we manually examine the gadget flow and craft a matching payload for the specific gadget. We then revisit the page, and using the automatically generated markup as described in §5.4, check where the attacker-controlled data appears. We use a proxy to insert the generated markup into the HTTP response, simulating the initial markup injection. For script execution and markup injection gadgets, we craft a payload with the appropriate break-out sequence to gain client-side script execution when the gadget re-inserts our data back into the DOM. For request, object, link, and WebSocket gadgets, we examine the outgoing network traffic, searching for the benign payload of the automatically generated markup. To not inflict any harm on the web service, we manually examine the code to verify that an attacker can control relevant parts of the URL or message content and can exploit it for one of the attacks listed in Table 3.

6 Empirical Evaluation

We now answer RQ2 and RQ3, outlined in §3, by conducting an end-to-end, large-scale evaluation of DOM gadget prevalence and impact in the real world. We report the duration of each step during the large-scale DOM gadget crawl and analysis in Table 4.

6.1 Data Collection

In June 2024, we conducted a large-scale data collection effort using the crawling infrastructure detailed in §5.1 from an EU vantage

Table 4: Duration of each step of the DOM gadget analysis

Step	Duration	Comment
Crawling & Dynamic Analysis	2 months	5.3 minutes per domain
Static Analysis	2 months	3 hours per page
Verification of DAST results	4 days	16 minutes per page
Manual Analysis of results	8 days	5 minutes per flow

point. The process targeted the top 15K responsive domains from the Tranco list (ID: W88P9) [48], a widely-recognized ranking of websites [58]. From these domains, our crawling pipeline extracted 572 581 URLs, of which the data for 522 860 webpages were successfully collected, ensuring high data integrity. To maximize coverage, we repeated the data collection process for each failed URL up to three times.

The dataset encompasses 19M JavaScript scripts, with an average of 36 scripts per page. In terms of raw content, the total lines of JavaScript code (LoC) spanned an impressive 10.3 billion, underscoring the vast scale of the collected data and providing a robust foundation for studying DOM gadgets in the wild.

We conducted a script similarity analysis across the collected pages. We consider two pages to be similar if the SHA-256 hashes of their scripts are identical. Thereby, we identified 367 245 unique pages, highlighting the diversity within the dataset.

6.2 DOM Gadgets In the Wild

In total, we found 2.6M DOM gadgets across 364K webpages, and 9K distinct sites, as per methodology described in §5.2. Table 5 shows the prevalence of DOM gadgets across various source types. The table highlights differences in the distribution of gadget types across flows, pages, and domains. We encounter many dataflows repeatedly during dynamic analysis, because the gadget code runs repeatedly in a loop or event handler. Thus, we deduplicate the collected dataflows with the same URL, source function, DOM selector, and sink.

We observed that Request gadgets are the most frequent overall, with 1.2M flows appearing on 263K pages. They are also the most widespread on the web, appearing on 7.2K sites, indicating that DOM-controlled network requests are a core pattern across a broad range of sites, driven largely by instructions like `querySelectorAll` and `element.attribute`. Following closely, Object gadgets are also widely distributed across 250K pages, though with ~2x fewer instances. Finally, consistent with prior research [49], we observed that Code Execution gadgets maintain a moderate prevalence with 81K flows, remaining a critical security concern.

On the other side of the spectrum, WebSocket and Navigation gadgets are the least common. WebSocket gadgets, at 174K flows, appear only on 5.1K pages, mirroring their specialized role in real-time communication. Navigation gadgets are the least prevalent overall, with only 3.9K flows across 2.2K pages, showing that direct DOM-driven navigation changes are rare.

6.2.1 Gadget Verification. Our verification of dynamic flows shows that, in 357 982 cases (13.38% of all flows), the benign string payload has entirely reached its intended gadget. Among these verified gadgets, Request, Markup, and Link gadgets were the most prevalent, contributing 65%, 22.6%, and 11.6% of the cases, across 1025, 453, and 703 sites, respectively. In total, we verified at least one gadget in 14 345 web pages across 2259 sites (25% of all sites), which indicates the prevalence of potentially exploitable DOM gadgets.

For the verification of static flows, we randomly selected a subset of flows for each gadget type and manually verified them. To ensure diversity and reduce sampling bias, we selected one data flow per site. In total, we manually inspected 440 data flows, of which 126 flows (28.6%) were determined to be false positives. Our analysis reveals that the static pipeline effectively identifies vulnerable flows to Code Execution, Navigation, and Request gadgets with false positive rates of 10% (6/60), 15% (18/120), and 7% (4/60), respectively. However, Link and Object, and Markup gadgets have higher false positive rates of 54% (54/100) and 44% (44/100), respectively. There are a few reasons. In many cases, the source and sink were actually the same DOM element, which was simply being modified, but the static analysis flagged it as a vulnerable flow. Additionally, we observed that a few third-party code fragments, which included a false positive flow, appeared across many websites, amplifying the false positives. Finally, in other cases, DOM data was not directly assigned to a sink, but still influenced the sink indirectly, e.g., via conditions over tainted values. These patterns made it harder for the static analysis to distinguish between real and benign flows.

6.2.2 New Gadgets and Gadget Types. Our analysis revealed that 77.2% (276,583) of the verified DOM gadgets represent new gadget types while the remaining 22.8% represent previously known script gadget types. Some of the latter might be caused by unpatched websites employing the vulnerable JavaScript libraries covered by Lekies et al. [49]. Since we are the first to examine the new gadget classes, we can infer that the gadgets of the new types (77.2%) are novel vulnerabilities not found in prior work. Unlike traditional script gadgets, the new gadget types do not result in markup injection or code execution, highlighting a significantly broader attack surface than previously understood.

6.2.3 Contribution of Static and Dynamic Analysis. In total, static analysis identified 59 341 DOM gadgets across 20 688 pages, which is a rather small fraction of all the gadgets found in Table 3. This has several explanatory reasons. First, static analysis also analyzed fewer pages, i.e., ~10% of the pages analyzed dynamically, because performing static analysis by modeling a web page's JavaScript code as a graph and traversing it to identify vulnerable data flows incurs high computational costs [41]. As mentioned in §5.2.2, we limited static analysis to 10 unique pages per site to balance coverage and scalability. Secondly, we set a conservative depth threshold of $T = 30$ on backward graph traversal from sink to source nodes, as the number of possible data flow slices increases exponentially with traversal depth. Hence, our static approach may fail to detect very long data flows, which we aimed to detect via dynamic analysis. Despite these constraints, static analysis revealed valuable complementary insights. We observed that in 19 702 pages, static analysis could identify at least one DOM gadget flow that dynamic analysis missed. Specifically, static analysis detected at least one data flow to Code Execution, Markup, and Link gadgets in 5081, 11 065, and 4544 pages, respectively, which dynamic analysis missed. These findings highlight the prevalence of data flows that are not triggered during page load but may become active under specific runtime conditions.

Table 5: Dataflows from DOM sources into security-sensitive sinks.

	DOM Gadget Type								
DOM Source	**CodeExec**	**Markup**	**Request**	**WebSocket**	**Navigation**	**Object**	**Link**	**Total**	**Verified**
document.querySelectorAll	65,124	71,121	361,870	41,624	1,399	148,795	123,594	813,527	134,222
document.querySelector	3,398	73,940	403,750	68,733	876	147,303	54,605	752,605	128,639
document.getElementsByTagName	4,800	24,312	256,804	34,922	822	192,085	113,801	627,546	39,699
document.getElementById	5,452	29,060	169,511	17,095	718	52,655	17,376	291,867	21,518
document.getElementsByClassName	2,547	24,533	40,376	8,628	45	32,085	18,221	126,435	33,904
document.getElementsByTagNameNS	322	1,318	9,563	3,727	42	7,415	6,626	29,013	0
document.elementFromPoint	0	2	43	2	0	1,030	3	1,080	0
document.elementsFromPoint	0	0	10	0	0	25	2	37	0
Total	81,643	224,286	1,241,927	174,731	3,902	581,393	334,228	**2,642,110**	
Verified	301	81,098	232,904	0	0	1,990	41,689		**357,982**
Pages	17,845	109,463	263,946	5,137	2,231	250,596	156,394	364,487	14,345
Sites	1,486	5,347	7,276	367	377	7,523	5,610	**9,022**	**2,259**

6.3 Analysis of DOM Selectors

We analyze DOM query selectors to understand properties of markups attackers need to inject to be able to exploit DOM gadgets. One prerequisite to exploit a DOM gadget is that the selector is not constrained in a way that inhibits crafting a matching payload. For example, a selector that reads a URL from a `data-` attribute of a `div` and sets it as the `src` attribute of a newly created script tag. A simple selector like `div[data-src]` is rather easy to fulfill. If the selector has additional constraints, e.g., on the shape of the DOM, like in `div[id='scr-wrapper'] > div > div > div[data-src]`, exploitation requires insertion of a subtree in the correct position, which can be more difficult to achieve.

We used our extended version of Foxhound, as detailed in §5.2.1, to collect a comprehensive dataset of query selector strings. We relied on dynamic analysis because it enables the automatic capture of the exact selector strings used at runtime—many of which are dynamically constructed and not easily available in static analysis. To assess the complexity of the DOM selectors we encountered, we devised a complexity metric based on the *Selector Specificity* [6]. The browser computes a selector's specificity to resolve conflicts, if two or more selectors try to change the same attribute of an element. The specificity consists of three scores (A, B, C) which count the A) ID selectors, B) class and attribute selectors, and C) type and pseudo element selectors in the selector string. For example, `nav > a[data-x] > div#id` has one ID selector (`#id`), one class or attribute selector (`[data-x]`) and three type selectors, i.e., `nav`, `a`, and `div`. This results in a specificity of $(1, 1, 3)$. As each of the three scores counts constraints on the markup, we decided to simply sum A, B, and C to compute our complexity score.

We computed the complexity score for all selectors from the deduplicated data flows from Foxhound. This leaves us with 36 272 unique inputs over 1 591 979 recorded inputs. Some selectors occur extremely often, e.g., the most frequent selector query is `script[src*='otSDKStub']`, originating from the OneTrust Cookie Consent banner, with 355 851 occurrences. Parsing failed for 13 324 of those across 444 unique values. Reason for failure is that the API does not report errors, but if the selector is syntactically invalid, the return value is the same as if no match was found, i.e., `null`. In some cases, we observed that Foxhound truncated overly long selectors in an effort to conserve memory. Nevertheless, we were able to correctly compute the complexity score for 35 828 values that occurred 1 578 655 times (99%). The most complex selector we encountered is `.aside-menu>ul>li>ul>li>span , .aside-menu>ul>li>ul>li>a, .aside-menu>ul>li>ul>li>ul>li>span, .aside-menu>ul>li>ul>li>ul>li>a`, with a complexity score of 28. Here, a fairly deeply nested DOM structure is required to match the selector, which might be difficult to achieve. The average complexity is 1.80 and the median complexity 2. This shows that the majority of encountered DOM selectors do not pose heavy constraints on the markup required to abuse the DOM gadget.

6.4 Analysis of Sanitization Code Patterns

Starting from the DOM gadgets discovered via static analysis, we now analyze them, checking for the presence of input validation or sanitization checks. As a first step, we reviewed academic and non-academic literature [3, 8, 12, 13, 17, 20, 34, 39, 45] looking for code patterns related to sanitization and validation procedures. Then, we grouped the identified patterns by data types and operation types, resulting in seven categories, i.e., checks on Strings, checks on Numbers, checks on Boolean, checks on DOM nodes, regular expression operations, and sanitization operations. The categories and code patterns are presented in Table 1 of the supplementary material we published as part of the research artifact (§8.4). To increase confidence in the completeness of our discovered patterns, we manually examined 300 data flows in which none of these patterns were present, finding no new ones. As a second step, we analyzed the data flows. For this analysis, we focused on the static analysis data flows as static analysis allows us to precisely extract the program slice of a given data flow and apply pattern matching on the code. The table presents the total number of instances each sanitization operation matched on a data flow code.

Overall, our results indicate that the majority of DOM gadgets lack even simple sanitization or validation logic. From the 59 341 static data flows, 36 348 of them (61.25% of the total) do not have patterns related to the String, Regex, DOM, or Sanitizers classes, nor equality operations (i.e. ==, ===, !==). These patterns are typically associated with validation or sanitization operations, suggesting that the data flows lack effective data checking. Of these, 6161 of them (10.38% of the total) do not contain any of the sanitization and validation patterns.

6.5 Markup Injection In the Wild

We found 204K dataflows across 1.8K domains to inject HTML markups into webpages leveraging in-browser dynamic taint tracking. Table 6 shows the distribution of the dataflows across various sinks and injection points.

6.5.1 Verification and False Positives. Unfortunately, not all of the captured dataflows are exploitable for markup injection, as modifying the input string can sometimes prevent the dataflow from being triggered. To identify attacker-controllable dataflows, i.e., true markup injection vulnerabilities, we generated test payloads following the methodology described in §5.3 and tested them at runtime by monitoring the payload execution. Overall, we identified 4722 verified dataflows related to markup injection across 34K webpages in our dataset.

6.5.2 Injection Points for DOM Gadgets. Among 4722 markup injection vulnerabilities, 4379 cases are directly exploitable for XSS, as these websites neither properly sanitize untrusted user input nor implement CSP as a browser-based mitigation. Consequently, leveraging DOM gadgets to enable XSS on these sites becomes unnecessary. For the remaining 343 markup injection points, the DOM gadgets that we found in §6.2 are the only viable exploit.

6.6 Analysis of New Attack Techniques

We now quantify the contribution of our newly-proposed attack techniques of §4.3.

6.6.1 Contribution of New Attack Techniques. An existing markup injection can only be used to successfully trigger a DOM gadget if the injected markup has a chance of being selected by the DOM selector. We evaluate the order of the elements selected by the DOM gadgets covered in §6.2 and points of markup injection covered in §6.5. We extract the DOM selectors of the DOM gadgets as well as the XPath of the injected markup from the data flows collected by Foxhound. Our crawler collects the HTML snapshots of the page's DOM after loading and script execution. We evaluate the relative position of the elements by loading the HTML snapshot in a browser, evaluating the gadget selector and injection XPath, and comparing the positions of the returned elements using `compareDocumentPosition()`.

We compared 253K combinations of DOM gadget flows and markup injection flows to assess their relative positions in the DOM. In 34% of the cases, the injected markup appeared after the element selected by the gadget, while in 8% it appeared before. These findings indicate that, for at least 34% of the identified DOM gadgets, successful exploitation requires the novel techniques introduced in §4.3, as they can reorder elements in the DOM.

For 57% of the flow combinations, our analysis failed. Both the selector and the XPath do not return an element if there is no matching element in the DOM. In this case, we cannot compare their positions and our comparison fails. We manually examined why the elements could not be found. In the most frequent case, the tainted string was written to a newly created element not yet inserted into the DOM. Subsequently, the element, now containing the attacker-controlled markup, was inserted into the DOM. Since Foxhound records the XPath at the time of the injection, it recorded an XPath of an unattached element that cannot be evaluated against the snapshot of the DOM. The XPath for the markup injection failed in 39% of the cases, while the gadget selector failed in 7% of the cases.

6.6.2 Applicability of New Attack Techniques. The presented techniques can improve the odds of a successful attack by abusing particularities in the HTML parsing process. This of course requires that the attacker is able to inject these elements in the first place. To assess the likelihood of this, we tested whether several popular HTML sanitizers allow these tags to pass through. In case they do not already do so, we configured them to allow `data-x` attributes, to test with a common attribute. This data attribute serves as a placeholder for arbitrary `data-` attributes. The CCS 2025 [7] homepage, for example, uses `data-size=xl` to set font sizes, something one might allow a sanitizer to pass through. The results are provided in Table 7.

6.7 Gadget Exploitability

We now assess the exploitability of the DOM gadgets that we found in §6.2 as per methodology described in §5.4, i.e., 357K verified DOM gadgets across 2.5K sites. The existence of an injection point is a fundamental requirement to be able to exploit these gadgets. Unlike prior works [44, 49] that assume that such injection points always exist, our approach can find injection points automatically. After cross-referencing the set of potential markup injection dataflows and verified DOM gadgets (i.e., a DOM write operation followed by a DOM read instruction), we identified a total of 304,843 end-to-end dataflows across 1.8K websites. We then filter the results to include only confirmed markup injection vulnerabilities.

As a result, for 657 flows across 37 sites, we identified both a verified markup injection flow and a verified DOM gadget flow, together constituting an end-to-end vulnerability. Table 8 summarizes our findings.

We point out that, even in the absence of an injection point for a DOM gadget, websites may still be at risk. A significant portion of injection vulnerabilities on the web originates from third-party code, and a site that is not exploitable today could become such if a third-party script is updated to include a markup injection flaw. In such cases, an attacker could leverage an otherwise dormant DOM gadget to escalate the injection into a more impactful vulnerability.

6.7.1 Manual Analysis. We manually examine data flows for 100 domains to identify exploitations and causes for false positives. We randomly sample 100 domains from the set of domains with DOM gadget data flows. For each domain, we choose two pages at random from the crawling results and examine all data flows for these pages. For each data flow, we revisit the page and manually assess if the potentially vulnerable flow is still present. If the flow is still present, we exploit the DOM gadget by injecting matching markups into the HTTP response. The specific exploit depends on the type of sink that is present in the flow. For example, for a code execution gadget, we trigger an alert, whereas for asynchronous request gadgets, we trigger a request and demonstrate control over the URL. As a result of this process, we successfully created exploits for about 10% of the analyzed domains.

For 75% of domains, we observed at least one data flow that was not exploitable, e.g., due to sanitization. In other cases, we could

Table 6: Dataflows from web attacker sources into DOM sinks, representing markup injection vulnerabilities.

	Source							🌐			
Sink	**loc.href**	**loc.search**	**loc.hash**	**doc.URI**	**doc.ref**	**postMessage**	**win.name**	**Flows**	**Verified**	**Pages**	**Sites**
innerHTML	29,659	2,164	2,638	2,094	11,373	142,985	38	190,951	4,095	26,140	1,526
document.write	3,286	219	16	2,237	3,695	5	17	9,475	417	6,462	356
insertAdjacentHTML	1,369	72	1	1	162	488	0	2,093	208	1,679	73
document.writeln	318	0	0	0	148	0	0	466	1	343	16
outerHTML	282	3	0	0	5	2	0	292	1	214	12
iframe.srcdoc	145	0	36	0	254	0	0	435	0	299	31
element.before	99	0	0	0	0	0	0	99	0	99	3
element.after	79	25	0	0	68	9	0	181	0	115	6
Total	35,274	2,495	2,691	4,335	15,708	143,492	55	**204,050**	**4,722**	34,223	1,849

Table 7: Examined Sanitizing Libraries

Sanitizer	`html`	`body`	`table`
Google Caja	✗	✗	✓
js-xss	✗	✗	✓
sanitize-html	✗	✗	✓
DOMPurify	✓☰†	✓☰†	✓

†: DOMPurify blocks both `html` and body by default, but enabling `RETURN_DOM` allows body to pass through and enabling `WHOLE_DOCUMENT` enables both `html` and body to pass through.

Table 8: Summary of verified end-to-end DOM gadget flows.

Vulnerability	**Sink**	**Flows**	**Pages**	**Sites**
Markup Injection	innerHTML	77	13	6
	document.write	13	12	3
Request Forgery	fetch.url	27	7	5
	XMLHttpRequest.open(url)	24	14	8
	fetch.body	318	38	2
	XMLHttpRequest.send	55	45	6
Code Execution	iframe.src	6	6	4
Link	a.href	157	105	15
Total		657	177	37

not trigger the data flow from our inserted markup because of some condition hidden in minified JavaScript we could not resolve. For 32% of domains, we encountered at least one data flow that was no longer present during our manual visit of the page. This may be caused by changes to the website or because our crawler encountered different ads that contained vulnerable data flows. This problem may occur when analyzing websites with third-party code, as the loaded scripts are unstable between visits.

6.8 Case Studies

We now present a few manually vetted case studies of the confirmed vulnerabilities. We chose these DOM gadgets because they are comprehensive and representative. At the time of writing, the affected pages did not contain the required markup injection to exploit the gadgets. We describe the steps of the exploits that will be possible once a change to the website introduces an otherwise benign markup injection.

6.8.1 Request Gadget A. This DOM gadget shown in Listing 2 reads a URL from a data attribute and creates an asynchronous request to that URL. The gadget writes the content of the response to the DOM using the unsafe `prepend` method of jQuery. Thus,

```
1  $.ajax({
2      url: $(".options__page").attr("data-admin-uri"),
3      data: {
4          action: 'moove_gdpr_cookie'
5      },
6      success: function (data) {
7          $('.options__page').prepend(data);
8      },
9      error: function (data) {
10         console.log(data);
11     }});
```

Listing 2: Request gadget A

```
1  <div class='options__page'
2       data-admin-uri='https://attacker.com'>
```

Listing 3: Payload for request gadget A

an attacker that controls the URL can inject arbitrary HTML into the DOM, including malicious scripts. Thereby, this request gadget enables the same attacks as the previously known script gadgets, albeit with a different sink. The attack utilizes the ability to control the source of the content written to the DOM.

The attack works similarly to the example in Figure 1b. The attacker uses the markup injection vulnerability (e.g., an URL query parameter reflected into the DOM) to craft an URL that injects the attacker's markup into the DOM. They send the crafted URL including a payload that triggers the DOM gadget to the victim, who opens it. The victim's browser loads the website including the attacker-provided markup in Listing 3. The DOM gadget code from Listing 2 selects the attacker's markup because it matches the selector in line 2. The extracted URL points to the attacker's server, which returns an HTML response with a malicious script. The statement in line 7 writes the content of the response to the DOM, executing the malicious script.

6.8.2 Markup Injection Gadget B. This DOM gadget shown in Listing 4 rewrites every `<div>` element with class `bcembed` on the website and inserts a child element. The gadget reads the `data-bcid` attribute and writes the value as the `id` of the newly inserted `<video>` element. Because the newly inserted element is created insecurely by string concatenation (line 23-24), it is possible to break out of the `id` attribute and inject a malicious node. Since the gadget rewrites every matching `<div>` element (line 2), the position of the markup injection is not relevant. However, there are additional conditions for the vulnerable data flow: The element must match an additional class. While there are sanitization steps on the value that the gadget reads from the DOM, they only ensure that the value will be a valid ID but do not prevent exploitation. This gadget

```
// ...
e.find('div.bcembed').each((
      function (e, t) {
          waypoint_debug[e] = jQuery(t).waypoint(
            (
              function () {
                bc_loadplayer(jQuery(this.element));
                this.destroy()
              }));
        }))
// ...
function bc_loadplayer(e) {
  void 0 !== e.attr('data-bcid') &&
  e.attr('data-bcid', e.attr('data-bcid')
    .replace(/ /g, ''));
  n = !!e.hasClass('bcgallery');
  if ('live-iframe' == e.ptype || n )
    if (i) {
        e.vid = 'bcvid-'
          + e.attr('data-bcid').replace(/,/g, '-')
          + '-' + Math.floor(999999 * Math.random() + 2);
        e.html(
        '<div class="video-js"><video preload="auto" class="
            vjs-tech" id="'
          + e.vid + '"></video></div>'
        );
    // ...
```

Listing 4: Markup injection gadget B

```
<div class='bcembed bcgallery'
     data-bcid='testpayload"><img/src/onerror=alert(1)>'>
```

Listing 5: Payload for markup injection gadget B

is similar to the script gadgets covered by previous work [49] and can be used to inject malicious HTML wrapped in a benign `<div>` element that bypasses sanitization.

The attacker uses the markup injection to inject the markup shown in Listing 5. When the gadget code concatenates the content of the `data-bcid` attribute in lines 23-24, it breaks out of the id attribute and creates a new `<img>` with an `onerror` handler. Subsequently, the gadget writes the newly created element to the DOM, executing the malicious JavaScript code in the `onerror` handler.

6.9 Script Gadget Benchmark

To assess the false negative rate of our detection pipeline, we evaluate it on a benchmark of known script gadget vulnerabilities. We create a benchmark based on a publicly available collection of JavaScript libraries with script gadget vulnerabilities [2]. The collection contains 15 proofs of concept for script gadgets in popular frameworks. We discard two of the samples, Google Closure and jQuery, because both exploits rely on DOM clobbering.

From each of the remaining 13 libraries we create a benchmark sample as following: We create a simple website that includes the library scripts and additonally reflects the content of the URL fragment into the page, resulting in a client-side markup injection vulnerability. We copy the example code from the libraries documentation to create a realistic usage scenario of the library and its features. Afterwards, we crawl and analyze the resulting websites like any other website in §6.2 and test if our tool detects the vulnerabilities. Our toolchain detects 8 of the 13 script gadget vulnerabilities, resulting in a false negative rate of 38.5% We would have missed one of the benchmark samples during the large-scale crawl due to a race condition that has since been fixed.

7 Related Work

DOM-related vulnerabilities have been the subject of extensive research. One of the earliest and most well-known examples is DOM-based Cross-Site Scripting (DOM XSS) [50], where untrusted user input is written directly to the DOM, leading to arbitrary JavaScript execution. Robust defenses against traditional DOM XSS are well-established, relying on input validation [34, 61], or on restricting script execution through mechanisms like Content Security Policy (CSP) [72].

In recent years, however, the research community has shifted focus to attacks that bypass these defenses, primarily through the use of gadgets–code fragments that perform security-sensitive operations based on attacker-influenced data [30, 33, 38, 49, 53, 62, 63]. Gadget-based exploitation has become a recurring theme in JavaScript security, expanding the scope of DOM-related threats beyond traditional XSS. Recent studies have investigated gadgets across various attack surfaces.

Several works have focused on identifying gadgets that are triggered via prototype pollution [30, 38, 53, 62, 63]. GHunter [30] presented a runtime-based detection pipeline to uncover universal gadgets in JavaScript runtimes such as Node.js and Deno. By instrumenting the V8 engine with taint tracking, the authors detected dozens of previously unknown gadgets, including those leading to arbitrary code execution and privilege escalation, and provided a systematic evaluation of mitigations. Dasty [63] builds on dynamic taint analysis to analyze the server-side JavaScript ecosystem, identifying gadget flows in NPM packages. Silent Spring [62] explored the full attack chain from pollution sources to gadgets in Node.js applications, using a hybrid, static-dynamic detection approach, demonstrating the feasibility of end-to-end RCE exploits via polluted prototypes.

Other studies highlighted the diversity and complexity of prototype pollution gadget chains. Liu et al. [53] introduced a concolic execution framework to discover chained gadgets—where polluted properties influence other polluted flows—demonstrating more sophisticated forms of exploitation. On the client side, Kang et al. [37] proposed dynamic taint analysis to detect instances of attacker-controlled keys and values in property assignments, allowing attackers to add properties of the prototype object. Kang et al. [38] improved upon this technique, proposing GALA, a dynamic analysis framework that identifies gadgets by borrowing existing defined values on non-vulnerable websites and reusing them on victim ones where such values are undefined, thus guiding the property injection to flow to the gadget sink at runtime. The authors ran GALA on one million real-world websites, finding previously undetected gadgets in widely deployed frameworks such as Vue. These findings illustrate how benign DOM reads can be coerced into attacker-controlled flows.

Previous research also studied other types of gadgets that exploit characteristics of JavaScript execution environments to get triggered. For example, DOM Clobbering gadgets [28, 34, 44], initially proposed to bypass frame busters [59], transform innocuous-looking HTML markup into executable code by exploiting unexpected bindings between the DOM and JavaScript variables caused by naming collisions. Khodayari et al. [44] proposed a dynamic analysis approach to identify DOM Clobbering markups across a

wide range of mobile and desktop browsers, and fed the resulting markups into a hybrid detection methodology to detect clobberable gadgets in client-side JavaScript code. Heiderich et al. [33] uncovered mutation-based XSS (mXSS) attacks, demonstrating how certain DOM mutations performed by browsers, combined with insecure JavaScript patterns, can act as gadgets that turn initially safe HTML markup into executable code.

DOM clobbering is related but not similar to DOM gadgets or script gadgets, vulnerabilities caused by benign scripts on the website explicitly reading data from the DOM. In consequence, the challenges for creating matching markups and the employed techniques are different. A DOM clobbering markup must fulfill DOM constraints on nesting and named attributes. Liu et al. [52] proposed concolic execution to solve these constraints. The constraints on DOM gadgets markups are simpler; They consist of a selector that must be matched and an attribute holding the payload. We infer both based on dynamic or static data flows. However, most DOM selector methods select the first markup in the DOM, requiring us to devise techniques to move the attacker-injected markup inside the DOM.

Closely related to our work, Lekies et al. [49] introduced script gadgets. These are benign code snippets that attackers can repurpose to transform code-less input markup into code execution by abusing pre-existing DOM selectors, bypassing XSS mitigations like CSP [72]. Roth et al. [56] evaluated how script gadgets interact with deployed CSPs. Compared to prior works on script gadgets, we establish several new gadget types (e.g., Request Gadgets) based on new vulnerable sinks, perform an end-to-end analysis where we identify injection points instead of assuming an injection point exists, and propose new attack techniques for exploitation of DOM gadgets. Finally, we propose a methodology on how to detect and verify new gadgets automatically, compared to previous work's manual analysis. Taken together, these studies establish the significance of gadgets as a core abstraction for understanding exploitability in JavaScript programs. By systematically studying these vulnerable behaviours, we reveal a broader, underexplored class of gadget-based vulnerabilities grounded in the interaction between JavaScript logic and browser-managed DOM state.

8 Concluding Remarks

We summarize our findings and discuss their wider implications.

8.1 Takeaways

8.1.1 DOM gadgets are Ubiquitous. Data flows from the DOM into security-relevant sinks are prevalent on the web, with nearly 70% of the 522K examined webpages having DOM gadget flows, suggesting a high reliance on the DOM tree as a data source. Among the data consumed by gadgets, we have identifiers of additional web resources or full and partial URLs.

We find that 60% of the critical flows are neither sanitized nor validated, indicating that developers may not be aware of the risks posed by DOM reads should an attacker be able to perform a markup injection.

8.1.2 Script Gadgets Severe but not Prevalent. DOM gadgets broaden our understanding of critical data flows from the DOM. Whereas script gadgets focused on script execution as the primary attack goal, DOM gadgets extend the attack surface to include a broader range of vulnerabilities. Accordingly, the exploits differ as well: to exploit a script gadget, the attacker breaks out of the context into which the gadget inserts the attacker-controlled string, to gain script execution. For the other DOM gadget classes, the attacks are more subtle. If the DOM gadget selects data from the DOM to create the URL for a request, the attacker only needs to inject matching markup to control the URL.

Our analysis reveals that the new DOM gadget classes are prevalent both in number of flows (77%) and number of affected sites. The most prevalent are Request gadgets, closely followed by Link and Object gadgets. They occur more often than markup injection and code execution gadgets covered by related work [49, 56], with about 6× as many flows. Their consequences are diverse, ranging from information leakage to request forgery and code execution.

8.1.3 Complex Defense Landscape. Addressing the security risks of DOM gadgets is challenging. Currently, each gadget class requires its own mitigation strategy, without a universal solution.

The first line of defense for developers is input validation, which involves verifying that input strings contain expected values before using them in sensitive operations. For instance, developers should validate strings used by Network and WebSocket gadgets to ensure they are valid URLs. It is also important to note that these gadgets cannot be mitigated with traditional anti-CSRF solutions, such as anti-CSRF tokens. When a site relies on these tokens, the logic of adding them to outgoing requests is often part of the same gadget. Similarly, SameSite cookies provide only partial protection: While they mitigate cross-site requests, they are ineffective against same-site forgery attacks [43].

For other classes of gadgets, such as Markup Gadgets, custom input validation procedures may not be ideal, and developers might consider other solutions such as Trusted Types. Unfortunately, while Trusted Types are effective in practice, they are currently only available in Chrome, with Safari and Firefox still working on their implementations, which limits their overall impact. Additionally, Trusted Types are notoriously difficult to use in practice and do not offer comprehensive protection. For example, they are unlikely to provide any protection for JavaScript sinks [57]. A stronger defense may need to come from browser vendors. Similar to the push to escape < and > when serializing HTML [14], browsers could implement similar measures when reading properties, effectively preventing almost all DOM gadgets in the markup injection category, such as HTML-based script gadgets. Future work could assess the impact of this approach on performance and website functionality via field trials.

8.2 Threats to validity

We relied on web crawling to collect snapshots of web pages and their associated DOM gadgets. However, web crawling is a challenging task [64, 65], and our approach may have missed certain pages containing DOM gadgets, such as those hidden behind user authentication, requiring specific user interactions, or accessible only through particular web clients and vantage points. In addition, we limited static analysis to ten random pages of each site due to the large analysis time required by JAW (see, i.e., [41]). Furthermore, we

focused on DOM read APIs that can be controlled via query selectors. While alternative techniques such as DOM traversal through element relationships and XPath expressions are theoretically viable for reading DOM content, they tend to be highly brittle and sensitive to minor UI changes, making them significantly harder to exploit in practice. Therefore, we excluded them from our analysis. Consequently, our findings likely represent a lower-bound estimate of DOM gadget prevalence on the web.

We manually examined the intersection of pages with DOM gadgets and markup injection to validate the vulnerabilities. The DOM gadget exploits that we pass as payloads of the markup injection are more complex than regular XSS exploits, leading to more false negatives where the injected markup breaks during injection. Automatic validation to increase scalability must combine and solve the constraints on markups written via the markup injection and those imposed by the gadgets DOM selector. Future work could solve this problem with concolic execution similar to Symbolic DOM [52].

8.3 Ethical Considerations

We first describe how our experiment design minimizes the potential for harm and then touch on our disclosure process.

Harm Avoidance. We designed our experiment to avoid any harm to website operators and their visitors. Firstly, our crawler uses a round-robin crawling strategy, so that concurrent crawler instances do not visit the same domain at the same time. This minimizes the resource overhead caused by our experiment. We also transmit a header that identifies our crawler as a research project and contains an opt-out link. We also do not interact with the website, so we avoid accidentally interfering with the regular operation. Our crawler passively collects loaded scripts and taint flows. This happens purely on the client side, i.e., our machine.

We only test the exploitability of the client-side JavaScript code for the markup injection and DOM gadget exploitability experiments. This means we are the attacker and the victim at the same time, avoiding interference with other users. When evaluating the DOM gadget exploitability that triggers web requests, we only injected a benign string to check where it is inserted into the request. We examined the code to identify sanitization and encoding without sending any malicious requests to the server.

Vulnerability Disclosure. We are disclosing verified DOM gadgets to the affected site operators, per best practices for vulnerability notification [70]. We prioritize our reports by severity, focusing first on end-to-end exploitable data flows and websites with a known injection point. For any vulnerability explicitly mentioned in the paper, we anonymized the domain to reduce the risk of exploitation by malicious actors.

8.4 Open Science

In the spirit of open science, we publicly release all our artifacts[1]. This includes our crawling infrastructure to collect snapshots of webpages, static analyzer to detect DOM gadgets, tooling to identify and verify markup injection vulnerabilities, dynamic analysis scripts to verify the discovered DOM gadget data flows, and other evaluation scripts. We have integrated our improvements for DOM gadget detection into the main branch of the Foxhound project[2].

Acknowledgments

We gratefully acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972 as well as from the European Union's Horizon 2020 research and innovation programme under project TESTABLE, grant agreement No 101019206.

References

[1] 2018. *Client-Side CSRF.* https://www.facebook.com/notes/facebook-bug-bounty/client-side-csrf/2056804174333798/.
[2] 2023. *Google Security Research POCs.* https://github.com/google/security-research-pocs/tree/master/script-gadgets.
[3] 2024. js-xss: Sanitize untrusted HTML (to prevent XSS) with a configuration specified by a whitelist. (2024). https://github.com/leizongmin/js-xss.
[4] 2024. Link Manipulation. (2024). https://portswigger.net/kb/issues/00501003_link-manipulation-reflected.
[5] 2025. 13.2.6.1: Creating and inserting nodes. (2025). https://html.spec.whatwg.org/#creating-and-inserting-nodes.
[6] 2025. 17. Calculating a selector's specificity. (2025). https://www.w3.org/TR/selectors-4/#specificity-rules.
[7] 2025. ACM CCS 2025. (2025). https://www.sigsac.org/ccs/CCS2025/.
[8] 2025. Cross Site Scripting (XSS) Prevention Cheat Sheet. (2025). https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html.
[9] 2025. Cross-Site WebSocket Hijacking. https://portswigger.net/web-security/websockets/cross-site-websocket-hijacking
[10] 2025. Document: evaluate() method. (2025). https://developer.mozilla.org/en-US/docs/Web/API/Document/evaluate.
[11] 2025. *Document Object Model (DOM).* https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model.
[12] 2025. DOM Based XSS Prevention Cheat Sheet. (2025). https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html.
[13] 2025. Encoding API. (2025). https://developer.mozilla.org/en-US/docs/Web/API/Encoding_API.
[14] 2025. Escape "<" and ">" in attributes when serializing HTML. (2025). https://github.com/whatwg/html/issues/6235.
[15] 2025. EventTarget Interface. (2025). https://developer.mozilla.org/en-US/docs/Web/API/EventTarget.
[16] 2025. HTML Living Standard. (2025). https://html.spec.whatwg.org/.
[17] 2025. JavaScript Reference. (2025). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference.
[18] 2025. *Playwright browser automation framework.* https://playwright.dev/.
[19] 2025. *Project Foxhound.* https://github.com/SAP/project-foxhound.
[20] 2025. sanitize-html: Clean up user-submitted HTML, preserving whitelisted elements and attributes. (2025). https://github.com/apostrophecms/sanitize-html.
[21] 2025. Selectors Level 4. (2025). https://www.w3.org/TR/selectors-4/.
[22] 2025. The DOM Living Standard. (2025). https://dom.spec.whatwg.org/.
[23] 2025. W3C Standards and Drafts. (2025). https://www.w3.org/TR/.
[24] 2025. WHATWG Specifications. (2025). https://spec.whatwg.org/.
[25] Devdatta Akhawe, Adam Barth, Peifung E Lam, John Mitchell, and Dawn Song. 2010. Towards a formal foundation of web security. In *IEEE CSF.*
[26] Adam Barth, Collin Jackson, and John C. Mitchell. 2008. Robust defenses for cross-site request forgery. In *Proc. of the ACM Conference on Computer and Communications Security (CCS).*
[27] Souphiane Bensalim, David Klein, Thomas Barber, and Martin Johns. 2021. Talking About My Generation: Targeted DOM-Based XSS Exploit Generation Using Dynamic Data Flow Analysis. In *Proc. of the European Workshop on System Security (EUROSEC).* doi:10.1145/3447852.3458718
[28] Michał Bentkowski. 2019. XSS in GMail's AMP4Email via DOM Clobbering. (2019). https://research.securitum.com/xss-in-amp4email-dom-clobbering/.
[29] Frederik Braun, Mario Heiderich, and Daniel Vogelheim. 2024. HTML Sanitizer API, Section 4.2, DOM Clobbering. *W3C Draft Community Group Report* (2024). https://wicg.github.io/sanitizer-api/.
[30] Eric Cornelissen, Mikhail Shcherbakov, and Musard Balliu. 2024. {GHunter}: Universal Prototype Pollution Gadgets in {JavaScript} Runtimes. In *USENIX Security Symposium.*
[31] Jeremiah Grossman, Seth Fogie, Robert Hansen, Anton Rager, and Petko D Petkov. 2007. *XSS Attacks: Cross-Site Scripting Exploits and Defense.* Syngress.

[1] https://doi.org/10.5281/zenodo.16981621

[2] https://github.com/SAP/project-foxhound

[32] Chong Guan, Kun Sun, Zhan Wang, and WenTao Zhu. 2016. Privacy breach by exploiting postmessage in html5: Identification, evaluation, and countermeasure. In *Proc. of the ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. 629–640.

[33] Mario Heiderich, Jörg Schwenk, Tilman Frosch, Jonas Magazinius, and Edward Z Yang. 2013. mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[34] Mario Heiderich, Christopher Späth, and Jörg Schwenk. 2017. DOMPurify: Client-side protection against xss and markup injection. In *Proc. of the European Symposium on Research in Computer Security (ESORICS)*.

[35] Gareth Heyes. 2024. Using form hijacking to bypass CSP. (2024). https://portswigger.net/research/using-form-hijacking-to-bypass-csp.

[36] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. 2012. Remedying the Eval that Men Do. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[37] Zifeng Kang, Song Li, and Yinzhi Cao. 2022. Probe the Proto: Measuring Client-Side Prototype Pollution Vulnerabilities of One Million Real-world Websites.. In *Network and Distributed System Security Symposium (NDSS)*.

[38] Zifeng Kang, Muxi Lyu, Zhengyu Liu, Jianjia Yu, Runqi Fan, Song Li, and Yinzhi Cao. 2024. Follow My Flow: Unveiling Client-Side Prototype Pollution Gadgets from One Million Real-World Websites. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.

[39] Soheil Khodayari, Thomas Barber, and Giancarlo Pellegrino. 2024. The Great Request Robbery: An Empirical Study of Client-side Request Hijacking Vulnerabilities on the Web. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.

[40] Soheil Khodayari, Kai Glauber, and Giancarlo Pellegrino. 2025. Do (Not) Follow the White Rabbit: Challenging the Myth of Harmless Open Redirection. In *Network and Distributed System Security Symposium (NDSS)*.

[41] Soheil Khodayari, Kai Glauber, and Giancarlo Pellegrino. 2025. Do (Not) Follow the White Rabbit: Challenging the Myth of Harmless Open Redirection. (2025).

[42] Soheil Khodayari and Giancarlo Pellegrino. 2021. JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals. In *USENIX Security Symposium*.

[43] Soheil Khodayari and Giancarlo Pellegrino. 2022. The State of the SameSite: Studying the Usage, Effectiveness, and Adequacy of SameSite Cookies. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.

[44] Soheil Khodayari and Giancarlo Pellegrino. 2023. It's (DOM) Clobbering Time: Attack Techniques, Prevalence, and Defenses. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.

[45] David Klein, Thomas Barber, Souphiane Bensalim, Ben Stock, and Martin Johns. 2022. Hand Sanitizers in the Wild: A Large-scale Study of Custom JavaScript Sanitizer Functions. In *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*.

[46] David Klein and Martin Johns. 2024. Parse Me, Baby, One More Time: Bypassing HTML Sanitizer via Parsing Differentials. In *45th IEEE Symposium on Security and Privacy*. doi:10.1109/SP54263.2024.00092

[47] Lukas Knittel, Christian Mainka, Marcus Niemietz, Dominik Trevor Noß, and Jörg Schwenk. 2021. Xsinator. com: From a formal model to the automatic evaluation of cross-site leaks in web browsers. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[48] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Network and Distributed System Security Symposium (NDSS)*.

[49] Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo A Vela Nava, and Martin Johns. 2017. Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. In *CCS*.

[50] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[51] Xhelal Likaj, Soheil Khodayari, and Giancarlo Pellegrino. 2021. Where We Stand (or Fall): An Analysis of CSRF Defenses in Web Frameworks. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*.

[52] Theo Liu, Zhengyu amd Lee, Jianjia Yu, Zifeng Kang, and Yinzhi Cao. 2025. The DOMino Effect: Detecting and Exploiting DOM Clobbering Gadgets via Concolic Execution with Symbolic DOM . In *USENIX Security Symposium*.

[53] Zhengyu Liu, Kecheng An, and Yinzhi Cao. 2024. Undefined-oriented programming: Detecting and chaining prototype pollution gadgets in node. js template engines for malicious consequences. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.

[54] Wenbo Mei and Zhaohua Long. 2020. Research and Defense of Cross-Site WebSocket Hijacking Vulnerability. In *IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*.

[55] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting.. In *Network and Distributed System Security Symposium (NDSS)*.

[56] Sebastian Roth, Michael Backes, and Ben Stock. 2020. Assessing the impact of script gadgets on csp at scale. In *Proc. of the ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. 420–431.

[57] Sebastian Roth, Lea Gröber, Philipp Baus, Katharina Krombholz, and Ben Stock. 2024. Trust Me If You Can – How Usable Is Trusted Types In Practice?. In *USENIX Security Symposium*.

[58] Kimberly Ruth, Deepak Kumar, Brandon Wang, Luke Valenta, and Zakir Durumeric. 2022. Toppling top lists: Evaluating the accuracy of popular website lists. In *Internet Measurement Conference (IMC)*.

[59] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. 2010. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. (2010).

[60] Christian Schneider. 2019. Cross-Site WebSocket Hijacking (CSWSH). https://christian-schneider.net/CrossSiteWebSocketHijacking.html

[61] Theodoor Scholte, William Robertson, Davide Balzarotti, and Engin Kirda. 2012. Preventing input validation vulnerabilities in web applications through automated type analysis. In *IEEE Annual Computer Software and Applications Conference (COMPSAC)*.

[62] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. 2023. Silent spring: Prototype pollution leads to remote code execution in Node. js. In *USENIX Security Symposium*.

[63] Mikhail Shcherbakov, Paul Moosbrugger, and Musard Balliu. 2024. Unveiling the invisible: Detection and evaluation of prototype pollution gadgets with dynamic taint analysis. In *The Web Conference*.

[64] Aleksei Stafeev and Giancarlo Pellegrino. 2024. SoK: State of the Krawlers - Evaluating the Effectiveness of Crawling Algorithms for Web Security Measurements. In *USENIX Security Symposium*.

[65] Aleksei Stafeev, Tim Recktenwald, Gianluca De Stefano, Soheil Khodayari, and Giancarlo Pellegrino. 2024. YURASCANNER: Leveraging LLMs for Task-driven Web App Scanning. (2024).

[66] Cristian-Alexandru Staicu and Michael Pradel. 2019. Leaky images: Targeted privacy attacks in the web. In *USENIX Security Symposium*.

[67] Sid Stamm, Brandon Sterne, and Gervase Markham. 2010. Reining in the Web with Content Security Policy. In *The Web Conference*. 921–930.

[68] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. 2019. Don't Trust the Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In *Network and Distributed System Security Symposium (NDSS)*.

[69] Marius Steffens and Ben Stock. 2020. Pmforce: Systematically analyzing postmessage handlers at scale. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 493–505.

[70] Ben Stock, Giancarlo Pellegrino, Christian Rossow, Martin Johns, and Michael Backes. 2016. Hey, you have a problem: On the feasibility of large-scale web vulnerability notification. In *USENIX Security Symposium*.

[71] Avinash Sudhodanan, Soheil Khodayari, and Jaun Caballero. 2020. Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks. In *Network and Distributed System Security Symposium (NDSS)*.

[72] Mike West and Antonio Sartori. 2024. Content Security Policy Level 3. *W3C Working Draft* (2024). https://w3c.github.io/webappsec-csp/.

[73] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.