It's (DOM) Clobbering Time: Attack Techniques, Prevalence, and Defenses

Soheil Khodayari, Giancarlo Pellegrino
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
{*soheil.khodayari, pellegrino*}*@cispa.de*

***Abstract*—DOM Clobbering is a type of code-less injection attack where attackers insert a piece of non-script, seemingly benign HTML markup into a webpage and transform it to executable code by exploiting the unforeseen interactions between JavaScript code and the runtime environment. The attack techniques, browser behaviours, and vulnerable code patterns that enable DOM Clobbering has not been studied yet, and in this paper, we undertake one of the first evaluations of the state of DOM Clobbering on the Web platform. Starting with a comprehensive survey of existing literature and dynamic analysis of 19 different mobile and desktop browsers, we systematize DOM Clobbering attacks, uncovering 31.4K distinct markups that use five different techniques to unexpectedly overwrite JavaScript variables in at least one browser. Then, we use our systematization to identify and characterize program instructions that can be overwritten by DOM Clobbering, and use it to present *TheThing*, an automated system that detects clobberable data flows to security-sensitive instructions. We instantiate *TheThing* on the top of the Tranco top 5K sites, quantifying the prevalence and impact of DOM Clobbering in the wild. Our evaluation uncovers that DOM Clobbering vulnerabilities are ubiquitous, with a total of 9,467 vulnerable data flows across 491 affected sites, making it possible to mount arbitrary code execution, open redirections, or client-side request forgery attacks also against popular websites such as Fandom, Trello, Vimeo, TripAdvisor, WikiBooks and GitHub, that were not exploitable through the traditional attack vectors. Finally, in this paper, we also evaluate the robustness of the existing countermeasures, such as HTML sanitizers and Content Security Policy, against DOM Clobbering.**

***Index Terms*—DOM Clobbering, Attack Techniques, Prevalence, Defenses**

1. Introduction

Arbitrary client-side JavaScript execution has been one of the major threats against web applications since the early days, traditionally achieved by injecting JavaScript code into vulnerable pages, e.g., Cross-Site Scripting (XSS) attacks [1–11]. However, over the past 20 years, the growth of Web technology has introduced unforeseen interactions between JavaScript programs and the execution environment that can result in execution of arbitrary code without injecting JavaScript but only by injecting seemingly harmless HTML markups. The research community has only recently started studying the security of these interactions, mainly focusing on small code fragments called script gadgets [12] that react to injected HTML markups and transform it into code. Unfortunately, script gadgets are only the tip of the iceberg, and other complex interactions exist that attackers can abuse to hijack the program execution, which, to date, are largely unexplored.

DOM Clobbering is a vulnerability that originates from a naming collision between JavaScript variables and named HTML markups, where browsers replace pre-existing content of an undefined variable with an HTML element when the variable name and the element's `name` (or `id`) attribute match. Developers unaware of such behavior may use the content of undefined variables for sensitive operations, such as URLs for fetching remote content, and attackers can exploit it by injecting markups with colliding names. DOM Clobbering vulnerabilities have been known for over a decade, with the first instance identified in 2010 [13] where an `iframe` named `self` allowed attackers to overwrite the top window location of webpages containing frame-busting code, i.e., assignments such as `top.location = self.location`. Since then, security researchers have identified new, more subtle attack variants, combining pairs of HTML tags (e.g., [14, 15]) or browser-specific markups and attributes (e.g., [16–19]), and clobbering not only variables, but also deep object properties (e.g., [20–22]), nested window proxies (e.g., [21, 23]) and loops (e.g., [21]). When looking at the possible combinations of tags, attributes, code features, and runtime behaviors, prior works have merely scratched the attack surface, and, to date, we still miss a systematic and comprehensive exploration of this threat.

Recently, DOM Clobbering vulnerabilities in Gmail [22] and Google Analytics [24, 25] revamped new discussions about defenses, such as proposing to switch off named property accesses for DOM elements at the browser level (see, e.g., [25–27]), which has been dismissed since, according to Google Chrome telemetry data, about 10.5% of the pages in 2021 use named property accesses to implement functionalities that could otherwise break [28]. To date, the burden of protecting from DOM Clobbering attacks is solely on developers' hands, who can use existing countermeasures such as HTML sanitizers tailored to protect against DOM Clobbering, e.g., DOMPurify [7], or mitigate the risk of code execution via Content Security Policy (CSP) [29–31].

Unfortunately, DOMPurify protects only from specific DOM Clobbering cases, whereas CSP cannot prevent the execution of already-present code that reacts to markup injections, suggesting that existing countermeasures may be incomplete or even insufficient. As a last resort, developers can develop their own defenses, requiring a deep understanding of the main threat and its variants, which, unfortunately, may not be the case. For example, as witnessed by recent DOM Clobbering vulnerabilities discovered in HTML sanitizers, e.g., DOMPurify [17] and HTML Janitor [20], developers may still be largely unaware of the risk posed by DOM Clobbering vulnerabilities.

In this paper, we take a step back and take a deep look at DOM clobbering, with, to the best of our knowledge, the first systematic and comprehensive study on this neglected vulnerability class, covering three main aspects: a systematic exploration of the attack surface, a measurement of affected and vulnerable websites, and a review and evaluation of defenses. Starting from a comprehensive survey of prior DOM Clobbering vulnerabilities, we systematically generate candidate DOM Clobbering markups, and automatically test desktop and mobile browsers against them, covering all known HTML tags and attributes–including custom ones–and markup relationships. Then, we propose *TheThing*, a DOM Clobbering detection tool that combines hybrid program analysis, i.e., [32], for the discovery of potentially-vulnerable data flows, with forced execution, i.e., [33], for the automated vulnerability verification, leveraging the generated DOM Clobbering markups. We instantiate *TheThing* against the Tranco top 5K websites to quantify the prevalence and impact of DOM Clobbering vulnerabilities, processing, in total, over 24.6B lines of JavaScript code across 18.3M scripts and 205.6K webpages. Finally, we identify, review, and evaluate defenses, covering existing countermeasures and secure code patterns. In particular, we first precisely measure the cost-benefit trade-off of browser-level countermeasures and thoroughly test HTML sanitizers. Then, we review the vulnerable code discovered by *TheThing*, identify common developer mistakes, and distill a list of secure coding patterns.

Our results show that the attack surface of DOM Clobbering vulnerabilities is large, with only 481 out of 31,432 generated DOM Clobbering markups are currently known, and the remainings are either previously-unknown instances (148) or variants of known cases (30,803). When grouping markups by browser behaviors, we observe ten different behavioral groups, showing that while most of the attacks are shared across browsers, many others work with specific browsers only. In addition, our experiments discovered 114 new native browser APIs that these markups clobber in at least one browser, including security-sensitive APIs like cache storage [34] and trusted types [35]. Second, DOM Clobbering vulnerabilities are quite widespread, affecting 9.8% of the top 5K websites, including popular sites like GitHub, Fandom, Trello, Vimeo, TripAdvisor, WikiBooks and AliExpress, leading to severe consequences such as arbitrary code execution, client-side CSRF [32], and open redirections [36, 37]. Third, when looking at the browser-level defenses, disabling named property accesses can cause more breakage, i.e., 2,561 websites, than benefits, i.e., 491 vulnerable websites, with a cost-benefit ratio of 5.2:1 websites. In the absence of a browser-level fix, developers need to be particularly careful when choosing a countermeasure, as they balance protection with usability. For example, 55% of the most popular HTML sanitizers across the five most used web languages are vulnerable to at least one of the 31.4K clobbering markups by default. The remaining 45% sanitizers remove named properties, i.e., `id` and `name` attributes, which may interfere with the DOM manipulation operations. Also, our results show that CSP is insufficient because 85% of the discovered vulnerabilities can cause code execution without manipulating the `src` attribute. Finally, our results show that developers can fix vulnerabilities at the code level, and we identify eight distinct vulnerable code patterns to avoid and propose four secure patterns to fix them.

Listing 1: Example of DOM Clobbering vulnerability where named properties overshadow JavaScript variables.

```
var s = document.createElement('script');
let config =  window.globalConfig || {href: 'script.js'};
s.src = config.href;
document.body.appendChild(s);
```

Listing 2: Example of DOM Clobbering vulnerability where named properties overshadow native DOM APIs.

```
var s = document.createElement('script');
let b = document.documentElement.getAttribute('baseURI');
s.src = b + '/script.js';
document.body.appendChild(s);
```

Contributions. To summarize, this paper makes the following contributions:

- We conduct the first comprehensive and systematic study of DOM Clobbering, covering vulnerability, attack techniques, detection, prevalence, impact, and defenses.
- We propose a systematic technique to identify DOM Clobbering markups and test browsers automatically, identifying 148 previously-unknown ones, 30,803 new variants, and 114 new browser APIs that can be clobbered in at least one browser.
- We present *TheThing*, an automated detection tool for DOM Clobbering that uncovered 9,467 DOM Clobbering vulnerabilities, affecting 9.8% of the Tranco top 5K sites, of which 44 that we manually confirmed to be exploitable, including popular sites like GitHub, Fandom, Vimeo, Trello, TripAdvisor, and AliExpress.
- We evaluate the robustness of 29 client-side and server-side HTML sanitizers and CSP, showing that 55% of sanitizers are vulnerable and 85% of the DOM Clobbering vulnerabilities cannot be mitigated by CSP.
- We review existing countermeasures, analyze common mistakes of the 491 vulnerable sites, and distill a list of recommendations and secure coding patterns.

2. Background

Before presenting our study, we first dissect and introduce the DOM Clobbering vulnerability in §2.1, and then, we present the threat model of this work in §2.2.

2.1. DOM Clobbering Vulnerability

DOM Clobbering vulnerabilities originate from a naming collision between JavaScript variables and named HTML markups, i.e., markups with an `id` or `name` attribute [15, 38, 39]. When an undefined variable [40, 41] and an HTML markup have the same name, the browser replaces the pre-existing content of the variable with the DOM object mirroring the markup type. Listing 1 shows a snippet of vulnerable code, which loads a script whose URL is stored in a global configuration object, i.e., `window.globalConfig`. In more details, the code first creates a `script` tag (line 1), and then, it retrieves the global configuration object and stores it in a local variable `config` (line 2). If the configuration object does not exist, it uses a minimal default configuration, i.e., {`href: 'script.js'`} (line 2). Then, the program sets the `src` attribute of the newly created script tag to the `href` property of the configuration object (line 3) and appends the new script to the DOM tree (line 4).

The vulnerability originates in the assignment in line 2 because attackers can control the value of `window.globalConfig`, and ultimately, pick the script `src` value of their choosing by injecting an HTML tag with `id="globalConfig"`, e.g., `<a id="globalConfig" href="malicious.js">`. When parsing such a markup code, the browser maps the anchor tag element to the `window.globalConfig` property as mandated by the *named property access* rule of the HTML specifications (see [42–44]). The escalation to arbitrary code execution happens in line 3, when the code reads the `href` property of the object `window.globalConfig`, which no longer contains the object with the global configuration but it contains the attacker-controlled anchor tag whose `href` property value is `malicious.js`.

Attackers can abuse named property accesses in other ways, where instead of overwriting variables by HTML nodes, they can overshadow browser APIs. Listing 2 illustrates an example of such an attack. Similarly to Listing 1, this code also dynamically creates and loads a script. Instead of fetching the URL from a global configuration object, the code intends to use the `baseURI` attribute of the main HTML tag via the `document.documentElement` API (line 2). An attacker can manipulate the content of `src` in line 3 by shadowing the native property `document.documentElement` using an attacker-injected node in the DOM tree [45], e.g., a `form` element with `name="documentElement"` and the custom property `baseURI="malicious.js"`. When parsing the form tag, the browser maps the property `document.documentElement` to the JavaScript object representing the form tag (an instance of the `HTMLFormElement` class) which has a function called `getAttribute` which returns the value of the attribute `baseURI`, i.e., the string `malicious.js`.

2.2. Threat Model

In a DOM Clobbering attack, the attacker needs to insert an ad-hoc HTML payload into a target, vulnerable webpage. A *web* attacker [46, 47] can achieve that, e.g., adding a preview of a post to the client-side webpage by leveraging the URL parameters. Another example is the case where the attacker can implant a persistent DOM Clobbering payload in the target webpage, which can lie dormant, and exploited later on to attack a victim, e.g., adding persistent comments in the UI through Gmail's dynamic email feature [48] which allows including HTML content [22], or user-generated Markdown descriptions in code repositories that are turned into HTML content [49, 50]. Finally, a more powerful web attacker (e.g., [5, 12]) who is aware of a markup injection vulnerability can manipulate the DOM tree.

3. Problem Statement

This paper aims to answer the following questions:

(RQ1) DOM Clobbering Attack Techniques. When looking at the evolution of DOM Clobbering attack markups, we observe a consistent complexity growth, starting from a single HTML element [13] that can overwrite a variable, evolving with pairs of HTML tags [14, 15] that clobber properties of objects (2013-15), and then advancing into a wide variety of browser-specific combinations of different HTML tags and attributes that can not only overwrite variables, but also native DOM objects (2015-18) [16–19], nested object properties, and loop elements (2018-22) [21–23]. Despite the growth of markups' complexity, the exploration of the attack surface has not been conducted systematically, and to date, many of the possible combinations of tags, attributes, markup relationships and possible JavaScript object manipulations are not considered. As a first research question, we intend to fill this gap and exhaustively explore such an attack surface by generating clobbering markups and testing modern mobile and desktop browsers automatically.

(RQ2) Detection, Prevalence and Impact. While the existence of DOM Clobbering is known for more than a decade [13, 14], we still do not have a measurement about the prevalence, impact, and code patterns of this vulnerability. In this paper, we intend to quantify the prevalence of DOM Clobbering in the wild, identify vulnerable behaviours, and examine their impact to shed some light on possible causes and factors hampering web applications' security.

(RQ3) Defenses and Effectiveness. As a final question, we look at the defenses, their effectiveness, and cost-benefit, leveraging the data generated and collected from the previous answers, i.e., DOM Clobbering markups, vulnerability prevalence, and developer mistakes. In particular, we intend to re-evaluate the cost-benefit trade-off resulting from disabling named property accesses in browsers and thoroughly assess existing solutions such as HTML sanitization [7], Content-Security Policy (CSP) [22, 29], and freezing object properties [51] against DOM Clobbering. Finally, we intend to review developers' mistakes and identify vulnerable and secure coding patterns that can fix those issues.

4. Attack Techniques

The first part of this paper addresses RQ1, investigating the different ways DOM Clobbering markups can manipulate

JavaScript variables, object properties, and native APIs. Before presenting our findings (§4.2), we describe the methodology we followed to answer this RQ (§4.1).

4.1. Methodology

Our methodology comprises two main steps. First, we review existing works on DOM Clobbering attacks, looking for the various techniques to generate markups and at the browser specifications causing the overrides. Then, we apply the information gathered to generate markups exhaustively and thoroughly test browsers.

4.1.1. Systematization of Known Instances

As the first step, we systematically reviewed the existing literature on DOM Clobbering attack markups, i.e., the academic literature [7, 12, 13, 26], HackerOne vulnerability reports [52], the CVE database [53], Bugzilla bug reports [16], and non-academic resources (see, i.e., [14, 15, 17, 21–23, 39, 54, 55]). Then, for each discovered DOM Clobbering instance, we extracted the HTML tags, attributes, the clobbered target (e.g., variable or `window`/`document` property), the object type of the clobbered target (e.g., `HTMLElement` or `WindowProxy`), and tags relation (i.e., child, `srcdoc`, or sibling). Then, we looked for the corresponding browser specification rules that explain the reason why the clobbering instance works. When the rule defines other variants of the clobbering instance, we add them to the list of the instances. Accordingly, we reviewed the HTML and DOM specifications [56, 57], and GitHub issues in the specifications' repositories, i.e., W3C permissions policy [27], WICG document policy [19, 25], and WHATWG HTML and DOM standard repositories [18, 58]. Finally, we group instances together based on their similarity, i.e., tags, attributes, target, and the type of the value it refers to. Table 1 shows the result of our systematization.

4.1.2. Markup Generation and Browser Testing

Starting from our systematization, we derived a list of rules for generating DOM Clobbering markups, covering all HTML tags, attributes, tags' relations, and attack targets (i.e., a variable, an object property, or a native browser API). First, we generated candidate HTML markups for a target 'x' using all the 142 valid HTML tags, including a custom tag (e.g., `mytag`), and all the 244 valid HTML attributes, including a custom attribute. For each tag, we set the value of each attribute to 'x' and add the JavaScript code that checks whether the markup clobbers the target 'x'. Then, we generated markups for object properties '$x.y$' and '$x.x$' combining all pairs of the 142 HTML tags considering three relations: sibling tags, parent-child tags, and the `srcdoc` attribute value. The experiments with a single tag showed that only `name` and `id` attributes create named properties. Accordingly, to reduce the number of test cases to a testable size, the generation of markups for object properties did not consider combinations of all HTML attributes, but only those of the `name` and `id`, e.g., `id=x`, or `id=x, name=y`.

After generating all markups, we put each of them in a test webpage, along with a JavaScript code that verifies if the target is clobbered. Then, we instantiate each browser and visit the test pages automatically. For web browsers, we used BrowserStack [59] to programmatically control browser versions, names, and their execution life-cycle in a fully automatic fashion. We evaluated (the latest versions of) all mobile and desktop browsers available in BrowserStack (i.e., 16 browsers), and additionally tested the Tor Browser for the sake of completeness. Finally, for Safari, we considered three different versions that correspond to the three recent macOS operating systems as Safari cannot be upgraded standalone [60]. In total, we evaluated 19 browsers.

Overall, our generation algorithm produced 3,906,136 candidate test markups, of which 34,648 are for targets 'x', i.e., variables or native APIs, and the rest are for object properties '$x.y$' and '$x.x$'. When testing variables, we replace the target 'x' with the variable name generating in total 34,648 test cases for variables. When testing native DOM APIs, we replace the target 'x' with the API function or property name (e.g., the `cookie` property of `document`), obtaining 34,648 test cases per API function. As of October 2021, the total number of DOM API objects is 581 [61], of which 347 are `window` APIs (i.e., 291 properties and 56 methods) [62], and 234 APIs are for the `document` object (i.e., 178 properties and 56 methods) [63]. In total, we generated 20,130,488 test cases for native APIs.

4.2. Results

This section presents the results of our literature review and browser testing.

4.2.1. Systematization of Known Instances

Table 1 summarizes the DOM Clobbering markups. Our review identified 481 DOM Clobbering instances that we grouped into 13 classes based on their structural similarity. Each instance shows how a specific HTML markup (e.g., `<a id=x>`) can clobber a specific target, i.e., variable (e.g., x) or object property (e.g., window.x), and replaced it with a JavaScript object (e.g., x is shadowed by an `HTMLAnchorElement`). For each class, the table shows the clobbered target, the HTML code that can overwrite it, and the object type stored in the target. Also, the review of the HTML and DOM specifications resulted in the identification of five rules that instruct the browser to store the reference type in the target, which is mapped to each known DOM Clobbering instance. The rules are Named Access on Window ([56] §7.3.3), DOM Tree Accessors ([56] §3.1.5), Form Element ([56] §4.10.3), Iframe `srcdoc` attribute ([56] §4.8.5), and HTMLCollection ([57] §4.2.10.2), which we labeled as R1 to R5, respectively. The rest of this section details each group of clobbering markups and the rules abused by them.

Named Access Window. These group of markups leverage a single HTML element whose `id` or `name` is set to a target variable 'x', clobbering window.x due to browsers' compliance with the Named Access on the Window Object rule (R1) [42]. We reviewed this rule in §2.1. Note that we use window.x and 'x' interchangeably because all global

Name	Rule(s)	Target	Reference Type	Tag 1	Tag 2	Attribute 1	Attribute 2	Relation	Total	Reference
Named Access Window	R1	win.x, x	WindowProxy	iframe	-	name=x	-	-	1	[27, 42, 55]
		win.x, x	HTMLElement	TS1, TS2	-	name=x	-	-	5	[15, 17, 27, 42]
		win.x, x	HTMLElement	any	-	id=x	-	-	141	[14, 21, 22, 27, 42]
DOM Tree Accessors	R2	doc.x	WindowProxy	iframe	-	name=x	-	-	1	[21, 43, 54]
		doc.x	HTMLElement	TS1, TS2	-	name=x	-	-	5	[15, 43]
		doc.x	HTMLElement	object	-	id=x	-	-	1	[43]
		doc.x	HTMLElement	img, image	-	id=x, name=any	-	-	2	[15, 17, 43]
Form Parent-Child	R3, R1, R2	win.x.y, doc.x.y	HTMLElement	form	TS2, TS3	id=x ‖ name=x	id=y ‖ name=y	child	36	[15, 17, 18, 20, 21]
Nested Window Proxy	R4, R1, R2	win.x.y, doc.x.y	WindowProxy	iframe	iframe	name=x	name=y	`srcdoc` attr.	1	[21, 23, 54]
HTMLCollection	R5, R1, R2	win.x.x	HTMLCollection	any	any	id=x	id=x	child, sibling	141	[14, 22, 57]
		doc.x.x	HTMLCollection	TS2	TS2	id=x	id=x	child, sibling	3	[14, 43, 57]
		win.x.y	HTMLCollection	any	any	id=x, name=y	id=x	child, sibling	141	[14, 21, 22, 39, 54, 57]
		doc.x.y	HTMLCollection	TS2	TS2	id=x, name=y	id=x	child, sibling	3	[14, 43, 57]

Legend: R1= Named Access on Window Rule ([56] §7.3.3); R2= DOM Tree Accessors Rule ([56] §3.1.5); R3= Form Element Rule ([56] §4.10.3); R4= Iframe `srcdoc` Rule ([56] §4.8.5); R5= HTMLCollection Rule ([57] §4.2.10.2); win=window; doc=document; TS1=form, embed; TS2= object, img; image; TS3=button, fieldset, input, output, select, textarea.

TABLE 1: Overview of known DOM Clobbering markups grouped by their corresponding rules in the HTML [56] and DOM [57] specifications.

variables belong to the global `window` object by default.

DOM Tree Accessors. The markups of this group can shadow `document` properties because browsers comply with the DOM Tree Accessors rule (R2) [43], which instructs browsers how to retrieve properties of the `document` object (e.g., DOM elements). Similarly to the previous group, these markups use a single named HTML element (e.g., `object`, or `embed`) to clobber a property 'x' of the `document`.

Form Parent-Child Relationship. These markups clobber properties '$X.y$' where 'X' can be any of 'x', window.x, and document.x. First, they exploit either the rules R1 or R2 to clobber the base object 'X'. Then, they use the Form Element rule (R3) to clobber property 'y' of object 'X', i.e., the form elements' parent-child relationships where the browser creates a property of the second element for the first element's accessor variable [21]. DOM Clobbering code that rely on this technique comprise a `form` tag and a child (e.g., an `input`) named 'x' and 'y', respectively.

Nested Window Proxies. These markups use the Iframe `srcdoc` rule (R4) to create nested window proxies that are named with 'x' and 'y', respectively. Similarly to the previous group of markups, it uses the rule R1 or R2 to clobber the base object. Then, the stacked iframes enable attackers to exploit frame navigation features to clobber object properties like '$x.y$' [21, 23].

HTMLCollection. The last gour groups of markups rely on a different rule known as HTMLCollection (R5). Specifically, when two or more elements have the same `id` in the DOM tree, browsers create an array-like object called `HTMLCollection` [14, 64], which contains all elements with the same id. Elements inside HTMLCollections can be accessed by (i) their index in the collection and (ii) their `id` and `name`, enabling attackers to abuse R5 to clobber arrays [21] and loop elements (e.g., 'x' and '$x[i]$') as well as object properties like '$x.x$' and '$x.y$' [22]. Similarly to the previous techniques, rules R1-2 can be combined with R5 to clobber nested object properties like window.$x.y$.

4.2.2. Clobbering Variables and Object Properties

Our browser testing experiments uncovered 31,432 distinct DOM Clobbering markups that work in at least one browser, as summarized in Table 2, from which 145 clobber a variable 'x', and the remaining 31,287 clobber '$x.y$' and '$x.x$'.

Post-processing of Results. As the manual review of 31K individual instances is infeasible, we group instances by similar features. We start with preliminary groups based on the set of browsers they work in and the target they clobber. Then, we look at the structural features, i.e., tag1, tag2, attribute1, attribute2, and relationship, and we merge two groups when all the structural features but one are the same. Accordingly, we reduced the 31K instances to 74 classes, as shown in Table 2, and map each class to our systematization of known instances. In summary, out of the 74 classes, 10 classes rely on the Window Named Access, four classes on DOM Accessors, 13 classes on the Parent-Child Relationship, four classes on Nested Window Proxies, and finally 43 classes leverage HTMLCollections.

Findings. By comparing the 74 DOM Clobbering classes in Table 2 with the 13 previously identified classes in Table 1, we discovered that the 31,432 DOM Clobbering markups include 148 new instances, 481 previously known ones, and 30,803 variants of the known ones, which rely on one of the five DOM Clobbering techniques of §4.2.1.

The variants derive from markups that are already known for DOM Clobbering according to Table 1, but now have one or more additional attributes, or are permuted *in part* with a different HTML tag. For example, HTMLCollections clobbering `window` properties may be formed not only for two similar HTML tags as in Table 1 (e.g., two `a` tags with `id=x`), but also for certain combinations of dissimilar tags (e.g., `svg` and `a`), which accounts for a large number of the clobbering instances. Other variants are cases where additional `id` and `name` attributes are added to the existing clobbering markups. For example, when looking at `form` elements and their childern in Table 1, we observe that each tag of the markup has only one `id` or `name`. However, as demonstrated by the results in Table 2, these attributes may exist simultaneously on HTML tags and with similar or dissimilar values, resulting in additional clobbering variants.

In comparison, the new clobbering instances rely on new (pairs of) HTML tags and attributes that were previously not known to be applicable for DOM Clobbering. We observed that 28 out of the 74 identified classes contain at least one new instance, with a total of 148 new instances. From these, 22 classes contain only new instances (i.e., 142 instances). In the remaining of this section, we briefly describe the new

	Clobbered	HTML Markup: Tag 1	Tag 2	Attribute 1	Attribute 2	Relation	Total	New	Chrome 95.0.4638	Chrome 96.0	Chrome 92.0.4515	Firefox 94.1.2	Firefox 95.0	Firefox 39.0	Opera 65.2.3381	Opera 82.0.4227	Opera 3.2.3	Edge 95.0.1020	Edge 96.0.1054	Edge 95.0.1020	Safari 15.1	Safari 14.1	Safari 13.1	Safari 14.7.1	TB 11.0.1	SI 15.0.6	UC 13.3.8
Named Access Window																											
	win.x	TS2	-	id=x	-	-	106		●	●	○	●	●	○	●	●	○	●	●	○	○	○	○	○	●	●	●
⊕	win.x	customtag,iframe,TS5	-	id=x	-	-	8	1	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	win.x	TS6,bdi,bdo,big	-	id=x	-	-	6		●	●	○	●	●	○	●	●	○	●	●	○	○	○	●	○	●	●	●
	win.x	TS4,embed,form	-	n=x	-	-	5		●	●	○	●	●	○	●	●	○	●	●	○	○	○	○	○	●	●	●
	win.x	video,wbr,xmp	-	id=x	-	-	3		●	●	○	●	●	○	●	●	○	●	●	○	●	○	○	○	●	●	●
	win.x	aside,audio,b	-	id=x	-	-	3		●	●	○	●	●	○	●	●	○	●	●	○	○	●	●	○	●	●	●
⊕	win.x	applet	-	n=x	-	-	1	1	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
	win.x	iframe	-	n=x	-	-	1		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	win.x	base	-	id=x	-	-	1		●	●	○	●	●	○	●	●	○	●	●	○	○	●	●	○	●	●	●
	win.x	article	-	id=x	-	-	1		●	●	○	●	●	●	●	●	○	●	●	○	○	●	●	●	●	●	●
DOM Tree Accessors																											
	doc.x	TS4,embed,form	-	n=x	-	-	5		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
⊕	doc.x	applet	-	id=x ‖ n=x		-	2	2	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
⊕	doc.x	iframe	-	id=x ‖ n=x	-	-	2	1	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
	doc.x	object	-	id=x	-	-	1		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Form Parent-Child																											
	win.x.y	form	TS3,TS4 − fieldset	id=x ‖ n=x (& id=y)	id=y ‖ n=y	child	64		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	doc.x.y	form	TS3,TS4	n=x (& id=y)	id=y (& n=x ‖ n=y)	child	36		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	win.x.y	form	TS3,TS4	id=x (& n=y)	id=x & n=y	child	18		●	●	●	●	●	●	●	●	●	●	●	●	○	●	●	○	●	●	●
	win.x.y	form	TS3,TS4,embed	n=x	id=y & n=x	child	10		●	●	●	●	●	●	●	●	●	●	●	●	○	●	●	○	●	●	●
	doc.x.y	form	TS3,TS4,embed, form	n=x	id=y & n=x	child	10		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	win.x.x	form	TS3,TS4	id=x	id=x & n=y	child	9		○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○
	win.x.y	form	button	id=x ‖ n=x (& id=y)	id=y ‖ n=y	child	8		●	●	●	●	●	●	●	●	●	●	●	●	●	○	●	○	●	●	●
	win.x.x	form	TS3	n=x	id=y & n=x	child	6		●	●	●	●	●	○	●	●	○	●	●	●	○	●	○	○	●	●	●
	doc.x.x	form	TS3,TS4	n=x	id=y & n=x	child	6		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
⊕	doc.x.x, win.x.x	form	TS4,embed	n=x	id=y & n=x	child	4	1	○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○
⊕	doc.x.y	form	iframe	n=x	id=y & n=x	child	1	1	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
⊕	win.x.y, doc.x.y	form	TS4,embed	id=y & n=x	id=y & n=x	child	4	1	○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○
⊕	win.x.y	form	applet	n=x	id=y & n=x	child	1	1	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
Nested Window Proxy																											
	doc.x.x	iframe	iframe	n=x	id=y & n=x	srcdoc	1		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
⊕	doc.x.y	iframe	iframe	n=x	id=y ‖ n=y	srcdoc	2	1	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
	win.x.x	iframe	iframe	n=x	id=y & n=x	srcdoc	1		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
⊕	win.x.y	iframe	iframe	n=x	id=y ‖ n=y	srcdoc	2	1	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
HTMLCollection																											
	win.x.y	TS1,svg,customtag	TS1,plaintext	id=x	id=x & n=y	sibling	787		●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
	win.x.y	TS1,customtag − TS7,iframe	TS1,plaintext	id=x	id=x & n=y	child	774		●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
	win.x.y	abbr,dl,dt	TS13	id=x	id=x & n=y	child, sibling	274		●	●	●	○	○	●	●	●	●	●	●	●	●	●	○	●	○	●	●
	win.x.y	abbr,dl,image,img	TS8,TS12,TS20	id=x	id=x & n=y	child, sibling	392		●	●	●	○	○	●	●	●	○	●	●	●	●	●	●	●	○	●	●
	win.x.y	TS18	TS13,TS14	id=x	id=x & n=y	child, sibling	7,480		●	●	○	○	○	●	●	●	●	●	●	○	●	●	○	●	○	●	●
	win.x.y	address,dir,dt	TS15	id=x	id=x & n=y	child, sibling	338		●	●	●	○	○	●	●	●	●	●	●	●	●	●	○	●	○	●	●
	doc.x.y	applet	TS4,applet	id=x	id=x & n=y	child	4		○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
⊕	doc.x.y	TS4,applet,embed,form,iframe	TS4,applet,embed,form,iframe	n=x	id=y & n=x	sibling	13	13	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
⊕	doc.x.y	applet,embed,form,image,img	TS4,applet,embed,form,iframe	n=x	id=y & n=x	child	11	11	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
	doc.x.y	applet,object	TS4,applet	id=x	id=x & n=y	sibling	5		○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
	win.x.y	dir,div,dt,element	TS16	id=x	id=x & n=y	child, sibling	252		●	●	●	○	○	●	●	●	●	●	●	●	○	●	○	●	○	●	●
	win.x.y	div	TS17	id=x	id=x & n=y	child, sibling	66		●	●	●	○	○	●	●	●	●	●	●	●	○	●	●	●	○	●	●
	win.x.y	div,dl	TS12	id=x	id=x & n=y	child, sibling	186		●	●	●	○	○	○	●	●	●	●	●	●	●	●	●	●	○	●	●
	win.x.y	element,em,embed,fieldset	TS1,plaintext-iframe	id=x	id=x & n=y	child, sibling	876		●	●	●	○	○	●	●	●	●	●	●	●	○	●	○	○	○	●	●
⊕	win.x.y	embed	TS4,embed,form	n=x	id=y & n=x	child, sibling	10	10	●	●	●	●	●	●	●	●	●	●	●	●	○	●	○	○	●	●	●
⊕	doc.x.y	TS4,embed,form,iframe	TS4,embed,form,iframe	n=x	id=y & n=x	sibling	11	11	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
⊕	doc.x.y	TS4,embed,form	TS4,embed,form	n=x	id=y & n=x	sibling	25	25	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
⊕	doc.x.y	embed,image,img	iframe	n=x	id=y & n=x	child	3	3	●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
⊕	doc.x.y	embed,image,img	T3,TS4,embed,form	n=x	id=y & n=x	child	15	15	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	win.x.y	TS9,iframe	TS1,plaintext-iframe	id=x	id=x & n=y	sibling	1,436		●	●	●	○	○	●	●	●	●	●	●	●	○	●	●	○	○	●	●
	win.x.y	TS9	TS1,plaintext-iframe	id=x	id=x & n=y	child	1,301		●	●	●	○	○	●	●	●	●	●	●	●	○	●	●	○	○	●	●
⊕	win.x.y	form,image	TS4,embed,form	n=x	id=y & n=x	sibling	7	7	●	●	●	●	●	●	●	●	●	●	●	●	○	●	●	●	●	●	●
⊕	win.x.y	TS4,form	applet	n=x	id=y & n=x	sibling	4	4	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
⊕	win.x.y	image	embed,form	n=x	id=y & n=x	child	2	2	●	●	●	●	●	●	●	●	●	●	●	●	○	●	●	○	●	●	●
⊕	win.x.y	image,img	TS4,embed,form	n=x	id=y & n=x	child, sibling	16	16	●	●	●	●	●	●	●	●	○	●	●	●	●	●	●	●	●	●	●
⊕	win.x.y	TS4	applet	n=x	id=y & n=x	child	3	3	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
	win.x.y	ins	content,data	id=x	id=x & n=y	child, sibling	4		●	●	●	○	○	●	●	●	●	●	●	●	○	●	●	●	○	●	●
	win.x.y	TS7, TS8	TS1,plaintext − iframe	id=x	id=x & n=y	sibling	8,848		●	●	●	○	○	●	●	●	●	●	●	○	●	○	●	●	○	●	●
	win.x.y	TS8	TS1,TS11,plaintext − iframe	id=x	id=x & n=y	child	7,526		●	●	●	○	○	●	●	●	●	●	●	●	●	○	●	●	○	●	●
⊕	doc.x.x, win.x.x	object	TS4,embed,form	n=x	id=y & n=x	child	5	5	○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○
	doc.x.y	object	TS4	id=x	id=x & n=y	sibling	3		●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
⊕	doc.x.y	object	form,image,img	n=x	id=y & n=x	child	3	3	●	●	○	●	●	○	●	●	○	●	●	○	○	○	○	○	●	●	●
⊕	doc.x.y	object	iframe	n=x	id=y & n=x	child	1	1	●	●	○	○	○	○	●	●	○	●	●	○	○	○	○	○	○	○	○
	doc.x.y	object	image,img	id=x	id=x & n=y	child	2		●	●	○	○	○	○	●	●	○	●	●	○	○	○	○	○	○	●	●
⊕	doc.x.y	object	embed,object	n=x	id=y & n=x	child	2	2	○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○
⊕	win.x.y	object	TS4,embed,form	n=x	id=y & n=x	child, sibling	1	1	●	●	●	●	●	●	●	●	●	●	●	○	●	○	●	●	○	●	●
⊕	win.x.y, doc.x.y	object	TS4,embed,form	id=y & n=x	id=y & n=x	child	5	5	○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○
	win.x.y	svg	iframe	id=x & n=y	id=x & n=y	sibling	1		●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
	win.x.y	svg	TS1,plaintext	id=x & n=y	id=x & n=y	sibling	125		●	●	○	○	○	○	●	●	●	●	●	○	●	○	○	○	○	●	●
	win.x.y	svg,table	TS1,plaintext − TS19	id=x & n=y	id=x & n=y	child	157		●	●	○	○	○	○	●	●	●	●	●	●	●	○	○	○	○	●	●
	win.x.x	table	iframe	id=x	id=x & n=y	child	1		●	●	●	○	○	●	●	●	●	●	●	●	●	●	●	●	○	●	●
	win.x.x	table	TS1,plaintext,svg − TS10	id=x	id=x & n=y	child	119	0	●	●	●	○	○	○	●	●	○	●	●	○	○	●	○	○	○	●	●
	win.x.y	table	iframe	id=x & n=y	id=x & n=y	child	1		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Total							31,432	148	59	59	46	35	35	46	59	59	44	59	59	43	38	45	52	37	35	59	59

Legend: win=window; doc=document; TS_i= Tag Set in Table 7 of §A.2; n= name; (&p)= optional property p; − = minus operator; TB= Tor Browser; SI= Samsung Internet; UC= UC Browser; ● = clobbered; ○ = clobbering fails;

TABLE 2: Overview of DOM Clobbering markups. Rows marked with ⊕ are classes that contain new DOM Clobbering instances. For all rows, clobbering window.x also implies clobbering the variable x. Browsers with similar behaviours are grouped with the same color. The table highlights a total of 10 distinct groups of browser behaviours with respect to DOM Clobbering.

instances within each DOM Clobbering technique.

Named Access Window and DOM Tree Accessors. We discovered that any custom HTML tag (e.g., `customtag`) can be used to clobber a target variable x and window.x in all web browsers. Also, `iframe` tags with `id=x` can clobber document.x and named `applet` elements can clobber both window.x and document.x. In total, we found five new instances across four out of the 14 classes that rely on the Window Named Access and DOM Accessors techniques.

Form Parent-Child. We discovered that browsers like Firefox and Safari create accessor properties on JavaScript objects due to element's ancestral relationship in the DOM tree for previously unknown pairs of tags and attributes, such as a parent `form` tag with a `embed`, `iframe`, or `applet` child with both a `name` and `id` attribute. Overall, among the 13 classes that rely on elements' parent-child relationships, we found four new markups in four different classes.

Nested Window Proxy. We identified two new clobbering markups in two out of the four classes which use the Nested Window Proxies technique. In particular, we discovered that using the `id` attribute in the nested frames creates a named property on the base frame, referring to a `WindowProxy`, whereas `id` on the base frame does not create a `WindowProxy` accessible through the global `window` or `document`.

HTMLCollection. We found 137 new clobbering instances (across 18 classes) that lead to the construction of HTMLCollections in a different way. Specifically, we discovered that some browsers (e.g., Chrome and Firefox) create an HTMLCollection not only when two elements share the same `id`, but also when they have the same `name` value. However, we observed that this happens only for certain (combinations of) HTML tags, e.g., two `object` tags and two `form` tags with the same name can form an HTMLCollection, but not two `div` tags.

Analysis of Browsers' Behaviours. Our experiments revealed that browsers exhibit divergent behaviours when linking named HTML elements to JavaScript variables (Table 2). For example, we observed that for a significant fraction of the clobbering markups (i.e., 31,243 out of 31,432), there is at least one browser that disagrees with others, rendering the task of defending against DOM Clobbering increasingly more challenging. In summary, we identified 10 distinct groups of browser behaviours with respect to different DOM Clobbering markups, which are highlighted in Table 2 in colors, showing that while most of the attacks are shared across browsers, many others only work with specific browsers. The table shows that all Safari and iOS-based browsers have their own distinct behaviours, whereas browsers like Chrome, Opera, and Edge on Desktop and Android exhibit the same behaviour. Note that, in general, similarities in behaviours are expected because some browsers rely on the same underlying engine. For example, Chrome, Edge and Opera on Desktop are all Blink-based browsers [65], whereas iOS browsers are required to use the WebKit engine of Apple [66]. Finally, we observed that the least and highest amount of DOM Clobbering risk is associated with using browsers like Firefox Desktop/Android and Chromium-based browsers on Desktop/Android in which 35 and 59 classes of DOM Clobbering markups work, respectively.

4.2.3. Clobbering Native APIs

Overall, we identified a total of 347 DOM APIs (Table 12) that can be clobbered in at least one browser using one of the markups of §4.2.2, including 233 `document` and 114 `window` APIs. We observed that all `document` methods and properties except the `location` property (i.e., 233 APIs) can be clobbered in all browsers unanimously, as expected by the named property visibility algorithm [45] of the specification [19, 56]. However, this experiment resulted in a new finding that for a total of 114/347 `window` APIs (i.e., 91 properties and 23 methods), named properties can shadow native properties that would otherwise appear on the object in at least one browser, resulting in DOM Clobbering. This includes security-sensitive APIs such as the cache storage [34], notification API [67], trusted types [35], and web storage [68]–to name only a few instances. The complete list of clobbered `window` methods and properties is in Table 13 of §A.2. We observed that for 57/114 clobbered APIs, there is at least one browser that disagrees with others.

5. Detection and Prevalence

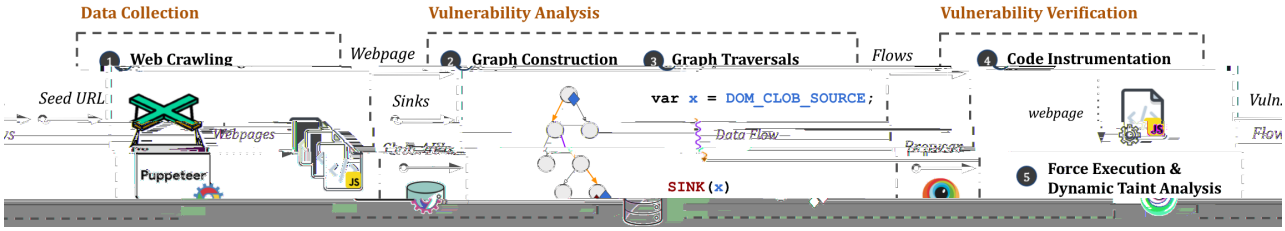
The second part of this paper intends to evaluate the impact, prevalence and variety of DOM Clobbering vulnerabilities in real-world web applications (RQ2 of §3). In §5.1, we first present *TheThing*, an automated DOM Clobbering detection tool. Then, in §5.2, we present our experiment results.

5.1. Detection

We formulate the problem of detecting DOM Clobbering vulnerabilities into a series of data flow analysis tasks where we identify *clobberable* JavaScript variables, object properties, and native APIs whose value ultimately reach security-sensitive instructions, such as script `src` and `eval`. Identifying such data flows via pure static analysis is not an easy task given the dynamic nature of client-side JavaScript programs [32, 69, 70] and the scale of the analysis as studying DOM Clobbering vulnerabilities requires the collection and analysis of hundreds of webpages of real web applications. Accordingly, we use and extend state-of-the-art property graphs for JavaScript and graph traversals [32] to identify potentially-vulnerable data flows and then use forced execution to confirm the presence of the vulnerability.

Figure 1 shows the architecture of *TheThing*. At a high level, it has three main components: (i) a web crawler to collect webpages' data and the JavaScript code, (ii) a vulnerability analysis component that uses property graphs and traversals for identifying potential DOM Clobbering sources and capturing data flows to security-sensitive sinks, and finally (iii) a vulnerability verification component that dynamically confirms the candidate data flows by instrumenting the code and forcefully executing it in a browser to check if the flow can occur at runtime. The rest of this section details each component.

Figure 1: Architecture of *TheThing*.



5.1.1. Data Collection

To collect the client-side code of web applications, we developed a JavaScript-enabled crawler leveraging Puppeteer [71] and Chrome DevTools Protocol (CDP) [72]. Starting from a seed URL of the website under test, it visits the webpages following a depth-first strategy, and stops when it doesn't find new URLs, or the maximum of 100 URLs is reached. During the visit, it collects the page resources (e.g., scripts) and runtime state values (i.e., fired events and DOM objects' properties) using the CDP and Puppeteer.

5.1.2. Vulnerability Analysis

Given the webpages' data collected by the crawler, *TheThing* creates a property graph of the client-side JavaScript program leveraging a modified engine of JAW [32]. Then, we formulate the problem of finding potential DOM Clobbering data flows into a series of graph traversal queries.

Hybrid Property Graphs. HPGs are graph-based representations of client-side JavaScript programs that unify multiple static code representations and runtime state values. State values are event traces and environment properties, e.g., the values of cookies and web storage. The static code representation comprises several graphs, e.g., Abstract Syntax Tree (AST), Control Flow Graph (CFG) and Program Dependence Graph (PDG) that model the nesting of the syntactical constructs of a program, the order and conditions for the execution of program instructions, and the data flow and control dependencies within the statements of a program, respectively. HPGs also model the event-driven transfer of control within JavaScript programs via the Event Registration, Dispatch and Dependency Graph [32]. Finally, they include Semantic Types, which are labels initially assigned to source and sink nodes to capture the semantic of those instructions and then propagated through the graph following the program calculation. These representations are encoded in a directed graph in which nodes and edges can have labels and key-value properties, known as a labeled property graph [32, 73].

Model Construction. After collecting the webpages' scripts and state values, *TheThing* instantiates an HPG, and imports it into a Neo4j docker instance [74], allowing the graph to be traversed declaratively using the Cypher query language [75]. Unfortunately, we could not use JAW as-is and modified it to address several of its shortcomings. First, when building a graph, JAW normalizes the webpage code by combining code inside script tags into a single script. However, identifying DOM Clobbering sources may require to distinguish the code across two different scripts due to JavaScript variable hoisting [41] and double-clobbering [17]. For example, a runtime error in one script causes the browser to stop parsing that script, and continue with parsing of the rest of the scripts. Hence, variables initialized in the first script are treated as undefined and can be a candidate DOM Clobbering source. Such runtime errors can be caused intentionally by attackers by a preliminary clobbering, e.g., clobbering a native DOM function that is invoked in a script shadows its value to an HTML element, which is not callable, leading to a runtime error (Cf. Table 3). Accordingly, we changed the normalization procedure to keep track of the script of origin for each AST node.

Second, the semantic types of JAW are tailored for client-side CSRF vulnerabilities and are not sufficient to model DOM clobbering. Accordingly, we added a new set of generic semantic types for DOM Clobbering sources (Cf. Tables 3 and 9) and security-sensitive JavaScript sinks (Cf. Table 8).

Third, JAW does not fully support ES6, resulting in imprecise control and data flow models. Accordingly, we applied several enhancements. For example, we added support to bind the function call arguments to their definition parameters when the code uses the ES6 Rest parameters [76] and the Spread operator [77] which improves the precision of the call graph and PDG edges. Also, we created bindings for the `this` object depending on the calling context [78], and the binding for the `arguments` object for non-arrow functions [79] to improve pointer analysis tasks.

Analysis Traversals. After construction of an HPG, we traverse it to identify DOM Clobbering source nodes in the graph. Table 3 presents the various types of DOM Clobbering sources and their properties. The table shows that clobberable native DOM APIs discovered in §4.2.3 can act as a DOM Clobbering source. Identifying these objects in the program is a matter of searching for a pre-defined syntactic structure, which is similar to other taint-style vulnerabilities like client-side XSS. However, contrary to the traditional taint analysis, not all DOM Clobbering sources are pre-defined syntactic objects. Instead, they can be a specific property of a program, identifying which requires tracking the propagation of data flows within the program itself. This is because any used variable that is undefined within its execution context (i.e., previously not declared and assigned) can act as a DOM Clobbering source. To identify such sources, we use PDG data dependency edges, which specify that a variable defined at a source node is subsequently used at the destination node. Specifically, we query the graph for Identifier nodes containing a variable v with no incoming PDG edge from any AssignmentExpression or VariableDeclaration nodes that

Object	DOM Clobbering Source When?
v	S1: $v \in$ NP, CLOB(v) S2: $v \notin$ NP, v and window.v are not assigned before, v is not declared with `var`, `let` and `const` before
window.v	S3: $v \in$ NP, CLOB(v) S4: $v \notin$ NP, v and window.v are not assigned before, v is not declared with `var` afterwards within the same script, or anywhere before S5: $v \notin$ NP, v or window.v is assigned or declared with any of the `var`, `let` and `const` keywords within any previous script that contains an invocation of function f such that $f \in$ NP, CLOB(f)
document.v	S6: $v \notin$ NP S7: $v \in$ NP, CLOB(v)

Legend: NP= native property; CLOB(v)= v is a clobberable NP based on §4.2.3.

TABLE 3: Description of properties of DOM Clobbering sources.

assign to or declare the variable v. If there is such PDG edge, we further check whether the declaration/ assignment statement can hinder the clobberability of v based on the criteria in Table 3, which can depend on the declaration scope (i.e., same script or not), declaration position (i.e., before or after), and the declaration keyword (e.g., `var` vs `let`) of that statement.

After identifying the source nodes, we associate to each of them a label that captures the semantic type of the source, e.g., a clobberable native property or custom variable (Cf. Table 9). Then, given a list of JavaScript sinks, we identify each of them in the graph and assign each a relevant semantic type. Semantic types assigned to sink instructions are propagated to other functions that encapsulate the same semantic, e.g., the type `WIN_LOC_WRITE` is set for instructions that set the value of `window.location`, such as `window.location.replace()`, and is then propagated to all other developer-defined functions that can set its value through one of their parameters. *TheThing* considers different sink types to enable us to capture the potential consequences of DOM Clobbering. The complete list of sinks is in Table 8, which is derived by surveying and aggregating the JavaScript sinks considered in prior academic and non-academic resources (see, i.e., [1, 5, 36, 37, 80–90]). Finally, we conduct forward data flow analysis by propagating semantic types from sources to sinks, and select those flows where a node with a sink semantic type is tainted with a source type (i.e., pick up the attacker-controlled values). The concrete queries are presented in Table 11 of §A.2. This component outputs a set of paths with potential data flows from a DOM Clobbering source to a sink.

5.1.3. Vulnerability Verification

Given a set of potential DOM Clobbering data flows, the goal of this step is to verify each flow and eliminate potential false positives. To accomplish this goal, *TheThing* features a light-weight, in-browser dynamic taint analysis engine leveraging *Iroh.js* [33]. After instrumenting the code with Iroh for dynamic analysis, we first check whether the source variable of the data flow is clobberable by creating a suitable HTML clobbering payload for that variable using the DOM Clobbering classes of §4. We inject the payload to the DOM tree and subsequently verify the clobberability of the source variable by dynamically logging its value at the source location.

Threat	# Sinks	# Flows	# Conf.	# Pages	# Sites
Client-side XSS	37,941,540	3,688	3,677	1,572	474
Request Forgery	2,555,147	1,406	1,403	541	398
Storage Manipulation	1,047,512	1,369	1,365	418	382
Open Redirect	1,306,603	1,228	1,227	391	385
JSON Injection	9,610,162	793	793	345	343
Cookie Manipulation	1,702,340	266	266	204	195
Websocket Hijacking	21,252	367	367	183	147
RegEx Injection	13,325,791	284	284	98	98
Doc. Domain Manip.	55,266	85	85	69	69
postMessage Manip.	119,971	0	0	0	0
File Read Path Manip.	57,789	0	0	0	0
Total	67,743,373	9,486	9,467	3,821	491

Legend: Conf.= Dynamically Confirmed

TABLE 4: Prevalence and impact of DOM Clobbering in Tranco top 5K sites. The table shows the number of clobberable data flows to security sensitive sinks of Table 8, the number of affected webpages, and websites.

As the next step, we confirm the existence of the data flow to the sink instructions. To do that, we first taint each clobberable source, execute the program by loading it via Puppeteer, and check if we can observe the data flow reported by the static analyzer. If that is not the case, we forcefully execute the path toward sinks to check if there is an execution of the program in which the data flow to the target sink occurs. We use forced execution to find candidate pages among those where Puppeteer could not connect sources with sinks, and later validate the presence of the vulnerability manually. Specifically, for each branch in the path control flow, we forcefully execute the program once for the true and once for the false branch, until we hit a execution path with the target data flow, or we exhaustively checked possible execution paths. We observed that the number of branches between DOM Clobbering sources and sinks is in practice small (i.e., less than 10), as we will show in §5.2. Finally, as forced execution may also lead to spurious execution paths, we manually validate the decision reported by *TheThing* and examine the exploitability.

5.2. Prevalence in the Wild

We quantified the prevalence and impact of DOM Clobbering on the top 5K websites using the Tranco list [91] of Nov 1st, 2021 (ID: Y3JG), where we first selected the top 5K domains by excluding the duplicates like local versions of websites (e.g., *google.com* vs *google.de*), and then instantiated *TheThing* for each of the them.

Data Collection Statistics. Starting from the 5K seed URLs, *TheThing* collected 205,696 webpages, ranging between 1 to 91 pages per site (41 pages on average). Out of the 205,696 webpages, 187,280 are unique pages based on their set of scripts. From the 187K pages, *TheThing* extracted 18,351,815 scripts with a total of 24,664,686,928 LoC. Accordingly, *TheThing* generated 187,280 HPGs by processing an average of 98 scripts and 131,700 LoC per page.

Vulnerability Prevalence. The analysis of 187,280 HPGs resulted in the identification of 20,580,350 DOM Clobbering sources and 67,743,373 sinks, which amounts to an average of 110 sources and 362 sinks per webpage. Out of these, static analysis revealed a total of 9,486 potential data flows from the sources to the sinks, from which the majority (i.e.,

9,467) were confirmed dynamically. We observed that these vulnerable data flows affect around 2% of the webpages (i.e., 3,821 out of 187,280) and 9.8% of the tested websites (i.e., 491 out of 5K) in total. Table 4 summarizes our findings.

Vulnerability Impact. We observed that the 9,467 vulnerabilities can have different security implications, as shown in Table 4. The most common consequence is XSS that accounts for around 38.8% of the vulnerabilities, whereas the least common consequence is document domain manipulation [83, 84] that corresponds to less than 1% of the total vulnerabilities. Other common consequences were client-side state manipulation (17.2%), client-side request forgery (14.8%) and DOM-based open redirection (12.9%). Finally, the remaining 15.3% of vulnerabilities had other repercussions like JSON injection and Websocket connection hijack. We provide more information on each of these threats in Table 8 of §A.2.

Verification and False Positives. Considering the high number of reported data flows by the static analyzer (Cf. §5.1.2), it was infeasible to verify all of them manually. Instead, we followed a semi-automatic approach leveraging a combination of dynamic analysis, forceful execution and manual analysis, as detailed in §5.1.3.

We observed that in a large number of cases (46.1%, i.e., 4,373 flows), the dynamic verification component can successfully confirm the existence of the vulnerability by loading the page and executing it via Puppetter, whereas in the remaining cases (i.e., 5,113 flows), it needs to force execute between one to ten conditional branches (four on average) before it can confirm or reject the data flow and terminate. As a result of this process, the verifier eliminated a total of 19 FPs across 11 of the 491 vulnerable sites, and confirmed the rest (i.e., 5,094 flows within 2,643 webpages of 491 sites). We manually verified and investigated the reason for each FP, and discovered that eight FPs occur during the data flow analysis for identification of DOM Clobbering sources, and 11 during the data flow analysis from sources to sinks. The former cases happened because a variable was declared or assigned using a dynamic code generation construct for which the statement nodes and PDG edges were missing in the HPG, and the latter cases occurred due to dynamically fetched code where the value of the tainted variables changed, inaccurate pointer analysis for dynamic property lookups, and removal of event handlers that changed the tainted variables.

Finally, we manually validated the feasibility of the forcefully executed data flows by randomly selecting two pages per site, from the 2,643 pages of the 491 websites whose data flows were confirmed by forced execution. Our random sampling included 491 sites, 982 pages and 2076 data flows, out of which we could not determine a realistic execution path for at least 42 data flows in 42 sites, leaving us with 2,034 vulnerable data flows of 491 websites.

5.3. Confirming Exploitability of Vulnerabilities

We manually examined whether the identified vulnerabilities can be effectively exploited by an attacker. Given the high number of affected webpages, we randomly selected two vulnerable pages per each of the 491 affected sites, and subsequently checked whether we can insert a DOM Clobbering markup in the page by leveraging the functionalities offered by the application, or through URL parameters, which could allow us to overwrite the clobberable variable identified by *TheThing*. To be able to use protected functionalities offered by the websites (e.g., creating posts, adding comments, etc) and also prevent any side effects for other users, we created our own test accounts for 358 sites that supported this feature without monetary costs, and for the rest, we limited our tests to the public functionalities (e.g., search) without persisting any data. As a result, we created a proof-of-concept exploit for 44 websites in total, affecting popular sites and functionalities like Trello boards, Wiki pages in WikiBooks and WikiDot, comments in Vimeo and VK, reviews in TripAdvisor and OpenTable, posts in Fandom and JustPaste, surveys in SuveryMonkey, poster designs in PosterMyWall, and finally item searches in GitHub Shop, AliExpress, AliBaba and Telam News–to name only a few examples. The exploits enable an attacker to achieve XSS, open redirect, and client-side request forgery in 35, five, and four sites, respectively. We refer interested readers to §A.1 for a few case studies of the confirmed attacks.

6. Defenses

This section addresses RQ3 of §3. First, in §6.1, we have a critical look at the existing countermeasures and evaluate their robustness and cost-benefit tradeoff leveraging what we learned from Sections 4 and 5. Then, in §6.2, we analyze the common mistakes of the 491 vulnerable sites (see §5), and distill a list of recommendations and secure coding patterns that can resolve those issues.

6.1. Evaluation of Existing Countermeasures

Disabling DOM Clobbering Features. DOM Clobbering can be solved by disabling named properties [19, 25, 27]. According to Chrome telemetry [28], disabling named properties for clobbered variable accesses could break ~10.5% of the *webpages*. Our results of §5.2 are in line with these numbers, and we observed that 13.3% of the webpages use at least an instance of clobbered variable accesses.

As webpages tend to reuse code via shared scripts, a patch in a script may fix multiple websites. Accordingly, using the number of webpages may not accurately quantify the cost of fixing breakage. As an alternative, we can measure the number of affected websites, and our results show that the affected pages do not concentrate on a small number of sites, but they scatter over 51.2% of the top 5K sites.

While breakage adequately measures the cost of this solution, it may not be a good indicator for the actual benefits, i.e., fixed websites. Our results show that 118 websites of 2,561 potentially broken sites will be fixed, which is about 4.61% of the broken websites (and 2.4% of the total). However, our results also show that a large fraction of vulnerable websites are not considered by breakage. In particular, we found 373 websites (76% of the vulnerable

HTML Sanitizer	★				Default	Strict	Bypassed	Pct.	Ref.
Client-side JS									
1. DOMPurify	8.7K	534	49.7K	7.9M	●	●	29,995	95.4%	[7]
2. Google Closure Lib.	4.3K	1K	-	117K	○	○	-	-	[92]
3. JS-XSS	4.4K	584	136K	8.7M	◐	◐	25,592	81.4%	[93]
4. Sanitize-HTML	2.8K	316	102K	4.7M	●	○	79	0.25%	[94]
5. Google Caja	1.1K	123	-	-	●	●	27,951	88.9%	[95]
Node.js									
1. Insane	394	21	-	55.3K	●	○	5	0.02%	[96]
2. Bleach	117	19	-	1.6K	●	●	2,288	7.2%	[97]
3. Angular-sanitize	100	237	49.1K	936K	○	○	-	-	[98]
4. Yahoo html-purify	40	6	-	708	●	●	28,807	91.6%	[99]
5. Arcgis	11	2	-	32.6K	○	○	-	-	[100]
Python									
1. Mozilla Bleach	2.3K	230	155K	17.5M	◐	◐	31,132	99.05%	[101]
2. LXML	2K	481	216K	29.9M	●	●	28,211	89.7%	[102]
3. HTML Sanitizer	61	19	-	17.9K	●	●	332	1.06%	[103]
4. Htmllaundry	27	4	-	1.1K	●	●	1,460	4.6%	[104]
5. Django-html-sanitizer	20	62	-	2.8K	○	○	-	-	[105]
PHP									
1. Htmlpurifier	2.4K	284	82.7K	2.5M	○	○	-	-	[106]
2. Html-sanitizer	333	36	-	30.8K	○	○	-	-	[107]
3. Symfony Sanitizer	104	1	-	7	○	○	-	-	[108]
4. HTMLawed	30	14	-	390K	●	●	21,211	67.4%	[109]
5. Typo3 Sanitizer	13	10	-	88.9K	◐	◐	23,942	76.1%	[110]
C#									
1. AntiXssEncoder	2.6K	1K	-	6.4K	◐	◐	31,390	99.8%	[111]
2. HtmlSanitizer	1.1K	162	1.8K	108K	●	○	654	2.08%	[112]
3. AJAX Toolkit	275	133	4.2K	264	○	○	-	-	[113]
4. NSoup	147	46	-	72	○	○	-	-	[114]
5. HtmlRuleSanitizer	50	16	30	308	○	○	-	-	[115]
Java									
1. Jsoup	9.2K	2K	98.4K	-	○	○	-	-	[116]
2. OWASP HTML Sanitizer	647	171	-	-	○	○	-	-	[117]
3. Antisamy	105	72	-	-	○	○	-	-	[118]
4. HtmlCleaner	-	-	-	824	●	●	28,951	92.1%	[119]
Total Vuln. (● + ◐)					16	13			

Legend: ★= GitHub Stars; = GitHub Forks; = GitHub UsedBy; = Monthly Downloads; ● = Vulnerable; ◐ = Partially Vulnerable; ○ = Not Vulnerable

TABLE 5: Robustness of top five HTML sanitizers of web programming languages against the 31.4K DOM Clobbering instances of §4.2. The table shows the results for both the default and the most strict sanitizer configurations. The tested sanitizer versions are in Table 10.

ones and 7.5% of the total) that will benefit from such a solution. Overall, when comparing the cost and benefits, the ratio of vulnerable over potentially-broken websites is about 1:5.2 (i.e., 491 vulnerable and 2,561 potentially-broken sites).

HTML Sanitization. HTML sanitizers can sanitize the input markups before adding them to the DOM tree, e.g., by removing the *id* and *name* attributes from certain (combinations of) HTML tags (Cf. §4). To assess the robustness of the popular HTML sanitizers against DOM Clobbering, we dynamically tested them against all of the DOM Clobbering instances we identified in §4. First, we selected the top five *web* programming languages based on the GitHub 2021 Octoverse report [120], i.e., JavaScript, Python, Java, C# and PHP. We considered both client-side and server-side JavaScript (i.e., node.js). Then, we searched for sanitizers of each language and selected the top five based on their GitHub stars, forks and UsedBy, and the number of downloads in their respective package managers (e.g., npm for node.js, packagist for PHP, etc). This process led to the identification of 29 HTML sanitizer libraries, as for Java, we identified only four sanitizers.

After identifying the popular sanitizers, we input the 31.4K DOM Clobbering markups identified in §4 to each of them, and for each input vetted whether the sanitizer removes or changes the named properties in the output markup. For each sanitizer, we tested both the default and most strict configuration that it offers. We marked a sanitizer as vulnerable if there is at least one clobbering markup that bypasses the sanitizer without being altered. Finally, we marked sanitizers as *partially* vulnerable when they encode the $<$ and $>$ symbols of HTML tags but do not remove or change the DOM Clobbering named properties because encoding these symbols would not help when applications expect inputs in an HTML format.

Table 5 summarizes our findings. In total, we observed that 16 and 13 out of 29 sanitizers are vulnerable to at least one DOM Clobbering markup in their default and most strict sanitization configuration, respectively. In both of the configurations, four sanitizers are only partially vulnerable, as they escape the markup rather than cleansing the named properties. Finally, when looking at the remaining 13 sanitizers, we observe that they implement a robust, enabled-by-default defense. However, in all cases, they remove named properties unconditionally, i.e., for *all* input markups including those combinations that do not lead to DOM Clobbering, e.g., an anchor tag with `name=x` does not clobber the variable x. While such a strict approach is effective, it may hinder the usability of these libraries in cases where developers need to use `id` and `name` attributes for legitimate functionalities.

Content-Security Policy (CSP). When attackers can clobber the `src` attribute of dynamically created scripts, they can load and execute arbitrary JavaScript code. In these cases, the CSP `script-src` directive [29] can be used to constrain the value of script sources to a set of trusted domains, preventing attacker-loaded code to be executed [12, 22, 30]. However, unlike malicious JavaScript injected by the attacker, injected HTML code is not blocked by CSP. Accordingly, CSP does not mitigate other variants of DOM Clobbering that do not require script `src` manipulation, e.g., clobbering the parameters of dynamic code evaluation constructs like `new Function()` can lead to CSP-bypassable XSS. Our evaluation in §5.2 shows that 37.7% of the DOM Clobbering vulnerabilities that lead to XSS (i.e., 1,385 out of 3,677), which accounts for 14.7% of the total vulnerabilities can be mitigated by CSP, whereas the remaining ones cannot.

Freezing Object Properties. Another way to mitigate DOM Clobbering is to freeze DOM objects [51], e.g., via `Object.freeze()` method [121], which prevents the object to be overwritten by named DOM elements. While effective, determining all objects and object properties that need to be frozen is a non-trivial, error-prone task for web developers. Also, sealed objects cannot be changed anymore, hindering the dynamic composition of webpages. Finally, native properties cannot be frozen, rendering this approach ineffective when the DOM Clobbering source is a clobberable native property, which accounts for ~21.5% of vulnerabilities (i.e., 2,037 out of 9,467) in §5.2.

#	Code Pattern	Description	# Flows	# Pages	# Sites
A	`var VAR2 = window.VAR1 || CONST;` `SINK(VAR2);`	`VAR1` is not declared or assigned yet, thus `window.VAR1` is clobberable.	3,134	1,214	143
B	`var VAR2 = [WinDoc.]BA || CONST;` `SINK(VAR2);`	`BA` is a clobberable built-in API (§4.2.3), thus `BA`, `window.BA` and `document.BA` are clobberable.	2,037	832	99
C	`[document.VAR1 = CONST];` `SINK(document.VAR1 || CONST);`	Assignment to `document` properties is always shadowed by DOM Clobbering (§4.2.3).	1,896	655	81
D	`let VAR1 = VAR2 = CONST;` `SINK(window.VAR1 || CONST);`	`VAR1` is declared with `let` that does not create property on window, thus `window.VAR1` is clobberable.	367	153	18
E	`SINK(window.VAR1 || CONST);` `VAR1 = CONST;`	`VAR1` is initialized without `var` in the same script and after the sink, but this does not result in *hoisting*.	1,635	792	116
F	`SINK(window.VAR1 || CONST);` `var VAR1 = CONST;`	`VAR1` is initialized with `var`, but in a different script and after the sink statement.	121	50	12

G	`BA() // clobberable built-in API`
	`[window.]VAR1 = CONST;`
	`SINK(window.VAR1)`
		`VAR1` is initialized in a script where a built-in method can be clobbered and cause an error in parsing that script, hence `window.VAR1` can be clobbered in a subsequent script (double clobbering).	53	36	7

H	`SINK(window.VAR1 || CONST);` `[window.]VAR1 = CONST;`	`VAR1` is initialized in a different script as a property of the window or without any modifiers after the sink statement, thus `window.VAR1` is clobberable.	224	89	15

Legend: BA= Built-in API; WinDoc = Window or Document Object; [*code*]= Alternative *code* statement; Red= Clobberable; Yellow = script 1; Orange = script 2;

TABLE 6: Overview of DOM Clobbering code patterns in the wild. Different background colors represent code in two different `script` tags.

6.2. Secure Code Patterns

Our evaluation of existing DOM Clobbering countermeasures in §6.1 revealed that they are not sufficient for complete protection in a large number of cases. In this section, we have a closer look at the variety of DOM Clobbering vulnerabilities in real web applications (§5.2), identifying vulnerable behaviours and the common types of coding mistakes. Then, we use these vulnerable behaviours to distill a list of recommendations and defensive coding patterns that developers could apply to prevent DOM Clobbering. To achieve this objective, we extracted the vulnerable lines of code and characterized them based on their high-level syntax and semantics, identifying eight distinct vulnerable code patterns in the wild.

Table 6 summarizes our findings. We observe that the most common mistakes are patterns A and E, in which the developer references an undefined variable through the `window` object, and then use the result in a sensitive instruction, whereas the least common, but also more complex mistakes are patterns F, G and H where the vulnerability originates due to the position of the instructions that span across two different script tags. Other common mistakes are patterns B and C, where developers treat custom and native `document` and `window` properties as trusted values that can be safely used in sensitive operations. The rest of this section presents secure coding patterns that can prevent DOM Clobbering.

Explicit Variable Declarations. As shown in Table 6, a key element enabling DOM Clobbering is use of the `||` operator to rely on specific defaults when the primary, intended variable or property is undefined. As an alternative solution, developers can initialize those variables with the default value when they are undefined using `var` declarations, which prevents named properties to overshadow them according to the named property visibility algorithm [45]. This solution could patch the patterns A, D, E, F, and H. When the value needs to be used in multiple scripts, as in patterns F and H, the declaration should be in the same (or a previous) script, but not in subsequent ones.

Strict Type Checking. Another common mistake enabling DOM Clobbering is treating DOM properties, like `document` and `window` properties as safe, trusted values (e.g., patterns B, C, and G). Instead, developers should extend the trust boundary to these properties, verifying their type before using them in security-sensitive instructions, e.g., using the `instanceof` [122] and `typeof` [123] operators.

Do Not Use Document for Global Variables. Properties of `document` can always be overwritten by DOM Clobbering, even immediately after they are assigned a value, as in pattern C. Accordingly, developers should refrain from using `document` as a mean to store and retrieve global values. Instead, they can declare variables with `const` or `var` in the global context, or use the `globalThis` object [124].

Namespace Isolation. While robust sanitizers in §6.1 remove named properties, an alternative solution is to separate the namespace of variables defined by JavaScript code and named properties in user-generated markups. For example, we observed that the markdown to HTML converter of applications like GitHub and BitBucket prefixes `id` and `name` attribute values of user-generated markup with a specific string. Motivated by this solution, one can monitor runtime changes in the DOM tree via the `MutationObserver` API [125], and prefix named properties of *all* dynamically inserted markups before adding them to the tree, which patches all patterns in Table 6.

7. Summary and Discussion

Clobbering Markups Come In Many Forms. In this paper, we proposed a systematic technique to identify DOM

Clobbering markups, and showed that they come in many forms, with a total of 31,432 attack markups that rely on five different techniques, including 148 new instances and 30,803 new variants. We observed that browsers exhibit divergent behaviours when handling named properties. For example, for a significant fraction of the markups (i.e., 99%), there is at least one browser that disagrees with others, making it increasingly more challenging to enforce robust defenses.

DOM Clobbering is Ubiquitous. DOM Clobbering vulnerabilities are prevalent, affecting over 9.8% of the top 5K sites, with the consequences ranging from XSS to user state manipulation, request forgery and client-side open redirects in the majority of the cases, i.e., 83.7% (see §5.2).

Defenses Helpful but May not Completely Cut it. The evaluation of existing DOM Clobbering countermeasures (§6) suggests that each can only mitigate a fraction of the attacks. For example, 55% of the popular HTML sanitizers are vulnerable to at least one of the 31K clobbering markups by default, and CSP cannot mitigate over 85% of the identified vulnerabilities. Protecting such a fraction of the attack surface without switching named properties off completely is a more costly task, requiring developers to be aware of corner case behaviors of browsers and revisit the design and implementation of their systems, e.g., strict type checking, explicit variable declarations, or namespace isolation.

Open Science and Website. To support the future research effort, we publicly release *TheThing* [126], the automated browser testing pipeline [127] that identifies clobbering markups (see §4), and an interactive version of markups[1].

Ethical Discussion. Our experiments on live sites do not target any real user. Tests requiring to persist data, e.g., store a markup, is exclusively restricted to user accounts that we created on those sites. Also, we excluded testing functionalities where we could not control the impact and visibility of the injected markup (e.g., publicly accessible posts and comments). Tests on public functionalities was performed without persistently injecting any markup.

The vulnerabilities and security risks identified in this paper affects 491 websites and 16 sanitizer libraries. We started the process of notifying the affected parties in March 2022 following the best disclosure practices [128, 129], where we prioritized our reports by severity. We sent an initial notification that includes the vulnerability details, or a proof-of-concept exploit, followed by an additional reminder every three weeks to maximize the remediation rate. At the time of preparing the camera-ready, we have notified all affected parties at least once, out of which 72 sites have already confirmed the issues, and 21 sites patched them, such as GitHub, Vimeo, Fandom, TripAdvisor and SuveryMonkey.

8. Related Work

Reusing the webpages' legitimate JavaScript code to obtain arbitrary client-side code execution have been the subject of several research endeavors in the past. Most notably, Lekies et. al. [12] described a new attack where small fragments of JavaScript code, known as *script gadgets*, are unexpectedly executed as a result of a non-script markup injected by attackers. The authors used a modified browser engine [1] to measure the prevalence of these gadgets, and demonstrated that they are prevalent and can bypass existing XSS mitigations, such as HTML sanitizers [7] and CSP [29, 30]. Later, Roth et. al. [130] quantified the impact of script gadgets on CSP in the wild. Similarly, Heiderich et. al. [6] discovered mutation-based XSS attacks (mXSS), showing how specific browser-based mutations of DOM content and insecure JavaScript that reads and rewrites HTML elements can transform initially secure DOM markup to code. While all these three attacks can transform non-script markup to executable code, the elements enabling DOM Clobbering is largely different, i.e., script gadgets rely on event handlers and mXSS attacks abuse `innerHTML` mutations, whereas DOM Clobbering is the result of a complex interplay of the default browser behaviors and insecure use of named property accesses in JavaScript programs. Contrary to these works, our study focuses on DOM Clobbering, systematically testing mobile and desktop browsers, identifying insecure coding patterns using both static and dynamic analysis techniques, and demonstrating their exploitability.

Multiple instances of DOM Clobbering vulnerabilities have been discovered in the last 12 years by both academics [7, 13, 131] and security analysts [14, 21–23, 54], with the first public instance identified in 2010 by Rydstedt et. al. as a way to circumvent frame busters [13]. The term 'DOM Clobbering' itself emerged in 2013, when Gareth Heyes [14] demonstrated how this class of vulnerabilities can escalate to client-side code execution. Due to such nefarious consequences of DOM Clobbering, prior academic studies has primarily focused on its defenses (e.g., [7, 26, 51]). Most notably, Heiderich et. al. proposed the JSAgents library [51] and later the DOMPurify sanitizer [7] to mitigate the security implications induced by markup injection, such as DOM Clobbering and client-side XSS [1, 5]. Our research completes the missing pieces of these works by systematically studying DOM Clobbering attack techniques, their prevalence, and effectiveness of the existing countermeasures.

9. Conclusion

In this paper, we performed, to the best of our knowledge, the first comprehensive study of DOM Clobbering, systematically covering clobbering techniques, browser behaviours, vulnerability prevalence, and defenses. Starting with a comprehensive survey of existing literature and dynamic analysis of 19 web browsers, we presented the first taxonomy of DOM Clobbering, uncovering 31K distinct markups that use five different techniques to clobber JavaScript variables. Then, we presented *TheThing*, the first DOM Clobbering detection tool, and instantiated it on the top of the Tranco top 5K sites, showing that DOM Clobbering vulnerabilities are prevalent. Finally, we demonstrated that existing countermeasures are not sufficient to mitigate a significant fraction of the vulnerabilities, and accordingly proposed several recommendations and secure coding patterns for developers.

1. https://soheilkhodayari.github.io/DOMClobbering

Acknowledgments

This work received funding from the European Union's Horizon 2020 research and innovation programme under the TESTABLE project (grant agreement 101019206).

References

[1] S. Lekies, B. Stock, and M. Johns, "25 million flows later: large-scale detection of DOM-based XSS," in *CCS*, 2013.

[2] Y. Nadji, P. Saxena, and D. Song, "Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense." in *NDSS*, 2009.

[3] J. Grossman, S. Fogie, R. Hansen, A. Rager, and P. D. Petkov, *XSS Attacks: Cross-Site Scripting Exploits and Defense*. Syngress, 2007.

[4] J. Dahse and T. Holz, "Static Detection of Second-Order Vulnerabilities in Web Applications," in *USENIX Security*, 2014.

[5] M. Steffens, C. Rossow, M. Johns, and B. Stock, "Don't Trust the Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild," in *NDSS*, 2019.

[6] M. Heiderich, J. Schwenk, T. Frosch, J. Magazinius, and E. Z. Yang, "mXSS Attacks: Attacking Well-secured Web Applications by Using innerHTML Mutations," in *CCS*, 2013.

[7] M. Heiderich, C. Späth, and J. Schwenk, "DOMPurify: Client-side Protection Against XSS and Markup Injection," in *ESORICS*, 2017.

[8] M. Samuel, P. Saxena, and D. Song, "Context-sensitive Auto-sanitization in Web Templating Languages Using Type Qualifiers," in *CCS*, 2011.

[9] P. Saxena, D. Molnar, and B. Livshits, "SCRIPTGARD: Automatic Context-sensitive Sanitization for Large-scale Legacy Web Applications," in *CCS*, 2011.

[10] D. Bates, A. Barth, and C. Jackson, "Regular Expressions Considered Harmful in Client-side XSS Filters," in *WWW*, 2010, pp. 91–100.

[11] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, "SWAP: Mitigating XSS attacks using a reverse proxy," in *ICSE Workshop on Software Engineering for Secure Systems*, 2009.

[12] S. Lekies, K. Kotowicz, S. Groß, E. A. Vela Nava, and M. Johns, "Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets," in *CCS*, 2017.

[13] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson, "Busting Frame Busting: A Study of Clickjacking Vulnerabilities at Popular Sites," *IEEE S&P*, 2010.

[14] G. Heyes, "DOM Clobbering," 2013, http://www.thespanner.co.uk/2013/05/16/dom-clobbering/.

[15] N. Jenkins, "Sanitising HTML – The DOM Clobbering Issue," 2015, https://fastmail.blog/advanced/sanitising-html-the-dom-clobbering-issue/.

[16] (2018) document.cookie DOM property can be clobbered using DOM node named cookie. https://bugzilla.mozilla.org/show_bug.cgi?id=1420032.

[17] (2015) Pentest-Report DOMPurify. https://cure53.de/pentest-report_dompurify.pdf.

[18] Provide an opt-out for inputs overriding form DOM API. https://github.com/whatwg/html/issues/2212.

[19] Feature Proposal: no [OverrideBuiltins]. https://github.com/WICG/document-policy/issues/16.

[20] (2018) Bypassing sanitization using DOM clobbering in HTML-Janitor. https://hackerone.com/reports/308158.

[21] G. Heyes, "DOM Clobbering strikes back," 2020, https://portswigger.net/research/dom-clobbering-strikes-back.

[22] M. Bentkowski, "XSS in GMail's AMP4Email via DOM Clobbering," 2019, https://research.securitum.com/xss-in-amp4email-dom-clobbering/.

[23] (2019) Clobbering the clobbered vol.2. https://terjanq.medium.com/clobbering-the-clobbered-vol-2-fb199ad7ec41.

[24] DOM Clobbering affecting Google Analytics script. https://twitter.com/zachleat/status/1387460811522813953.

[25] Feature proposal: Disable named access on window. https://github.com/WICG/document-policy/issues/32.

[26] A. Janc and M. West, "Oh, the Places You'll Go! Finding Our Way Back from the Web Platform's Ill-conceived Jaunts," in *IEEE EuroS&P Workshops*, 2020, pp. 673–680.

[27] Disabling DOM clobbering. https://github.com/w3c/webappsec-permissions-policy/issues/349.

[28] Chrome Platform Status: DOM Clobbered Variable Accessed. https://chromestatus.com/metrics/feature/timeline/popularity/1824.

[29] M. West, "Content Security Policy Level 3," *W3C Working Draft*, 2022, https://w3c.github.io/webappsec-csp/#directive-script-src.

[30] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock, "Complex security policy? a longitudinal analysis of deployed content security policies," in *NDSS*, 2020.

[31] S. Stamm, B. Sterne, and G. Markham, "Reining in the Web with Content Security Policy," in *WWW*, 2010, pp. 921–930.

[32] S. Khodayari and G. Pellegrino, "JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals," in *USENIX Security*, 2021.

[33] F. Maier. (2018) Iroh.js: Dynamic Code Analysis for JavaScript. https://maierfelix.github.io/Iroh/.

[34] The CacheStorage Web API. https://developer.mozilla.org/en-US/docs/Web/API/CacheStorage.

[35] K. Kotowicz, "Prevent DOM-based cross-site scripting vulnerabilities with Trusted Types," 2020, https://web.dev/trusted-types/.

[36] DOM-based open redirection. https://portswigger.net/web-security/dom-based/open-redirection.

[37] Z. Banach, "Open redirect vulnerabilities and how to avoid them," 2021, https://www.netsparker.com/blog/web-security/open-redirect-vulnerabilities-netsparker-pauls-security-weekly/.

[38] F. Braun, M. Heiderich, and D. Vogelheim, "HTML Sanitizer API, Section 4.2, DOM Clobbering," *W3C Draft Community Group Report*, 2021, https://wicg.github.io/sanitizer-api/#dom-clobbering.

[39] DOM clobbering. https://portswigger.net/web-security/dom-based/dom-clobbering.

[40] Undefined primitive type. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined.

[41] T. Rascia, "Understanding Variables, Scope, and Hoisting in JavaScriptt," 2021, https://www.digitalocean.com/community/tutorials/understanding-variables-scope-hoisting-in-javascript.

[42] HTML Living Standard: Named Access on the Window Object. https://html.spec.whatwg.org/multipage/window-object.html#named-access-on-the-window-object.

[43] HTML Living Standard: DOM Tree Accessors. https://html.spec.whatwg.org/multipage/dom.html#dom-tree-accessors.

[44] E. J. Etemad, T. A. Jr., T. Çelik, D. Glazman, I. Hickson, P. Linss, and J. Williams, "Selectors Level 4, W3C Working Draft," 2018.

[45] Web IDL Living Standard - Named Property Visibility Algorithm, Sections 3.4.7 and 3.9.7. https://webidl.spec.whatwg.org/#legacy-platform-object-abstract-ops.

[46] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song, "Towards a formal foundation of web security," in *IEEE CSF*, 2010.

[47] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *CCS*, 2008, pp. 75–88.

[48] "Dynamic email in Gmail becoming generally available on July 2019," 2019, https://workspaceupdates.googleblog.com/2019/06/dynamic-email-in-gmail-becoming-GA.html.

[49] J. Peek, "GitHub Handling of Named HTML Elements Generated by Repository Markdown Code," 2014, https://github.com/gjtorikian/html-pipeline/pull/111.

[50] V. Puzrin, "DOM Clobbering through Markdown Header anchors," 2015, https://github.com/markdown-it/markdown-it/issues/28.

[51] M. Heiderich, M. Niemietz, and J. Schwenk, "Waiting for CSP – Securing Legacy Web Applications with JSAgents," in *ESORICS*, 2015, pp. 23–42.

[52] DOM Clobbering Vulnerability Reports in HackerOne. https://hackerone.com/hacktivity?querystring=dom%20clobbering.

[53] DOM Clobbering Vulnerability Reports in Mitre. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=dom+clobbering.

[54] (2019) Clobbering the clobbered — Advanced DOM Clobbering. https://terjanq.medium.com/dom-clobbering-techniques-8443547ebe94.

[55] A. Nafeez, "DomFlow - Untangling the DOM for easy juicy bugs," 2015, https://www.blackhat.com/docs/us-15/materials/us-15-Nafeez-Dom-Flow-Untangling-The-DOM-For-More-Easy-Juicy-Bugs.pdf.

[56] HTML Living Standard. https://html.spec.whatwg.org/multipage/.

[57] DOM Living Standard. https://dom.spec.whatwg.org/.

[58] WHATWG DOM repository issues. https://github.com/whatwg/dom/issues.
[59] BrowserStack. https://www.browserstack.com/.
[60] S. H., "How to Update Safari without upgrading MacOS?" 2021, https://browserhow.com/how-to-update-safari-without-upgrading-macos/.
[61] Web APIs. https://developer.mozilla.org/en-US/docs/Web/API.
[62] The Window Interface. https://developer.mozilla.org/en-US/docs/Web/API/Window.
[63] The Document Interface. https://developer.mozilla.org/en-US/docs/Web/API/Document.
[64] (2021) The HTMLCollection Interface. https://developer.mozilla.org/en-US/docs/Web/API/HTMLCollection.
[65] The Blink Rendering Engine. https://www.chromium.org/blink/.
[66] H. Charlton, "Should Apple Continue to Ban Rival Browser Engines on iOS?" 2022, https://www.macrumors.com/2022/02/25/should-apple-ban-rival-browser-engines/.
[67] The Notification Web API. https://developer.mozilla.org/en-US/docs/Web/API/notification.
[68] The WebStorage API. https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API.
[69] S. H. Jensen, P. A. Jonsson, and A. Møller, "Remedying the Eval that Men Do," in *ISSTA*, 2012.
[70] S. Guarnieri and B. Livshits, "GULFSTREAM: Staged Static Analysis For Streaming JavaScript Applications," in *WebApps*, 2010.
[71] Puppeteer. https://github.com/puppeteer/puppeteer.
[72] Chrome devtools. https://chromedevtools.github.io/devtools-protocol.
[73] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in *IEEE S&P*, 2014.
[74] Neo4j. https://neo4j.com/.
[75] Cypher Query Language. https://neo4j.com/developer/cypher/.
[76] Rest Parameters. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters.
[77] Spread Operator Syntax. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax.
[78] S. Guarnieri and V. B. Livshits, "GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code," in *USENIX Security*, vol. 10, 2009, pp. 78–85.
[79] The arguments object. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments.
[80] DOM-based WebSocket-URL poisoning. https://portswigger.net/web-security/dom-based/websocket-url-poisoning.
[81] C. Polop, "Cross-site WebSocket hijacking," 2022, https://book.hacktricks.xyz/pentesting-web/cross-site-websocket-hijacking-cswsh.
[82] M. Steffens and B. Stock, "PMForce: Systematically Analyzing postMessage Handlers at Scale," in *CCS*, 2020, pp. 493–505.
[83] Dom-based document-domain manipulation. https://portswigger.net/web-security/dom-based/document-domain-manipulation.
[84] J. Schwenk, M. Niemietz, and C. Mainka, "Same-Origin Policy: Evaluation in Modern Browsers," in *USENIX Security*, 2017.
[85] T. A. Nideck, "What Are JSON Injections?" 2019, https://www.acunetix.com/blog/web-security-zone/what-are-json-injections.
[86] Client-side json injection. https://portswigger.net/kb/issues/00200370_client-side-json-injection-dom-based.
[87] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for JavaScript," in *IEEE S&P*, 2010, pp. 513–528.
[88] C.-A. Staicu and M. Pradel, "Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers," in *USENIX Security*, 2018, pp. 361–376.
[89] J. C. Davis, C. A. Coghlan, F. Servant, and D. Lee, "The impact of regular expression denial of service (redos) in practice: an empirical study at the ecosystem scale," in *ESEC/FSE*, 2018, pp. 246–256.
[90] DOM-based Local File-path Manipulation. https://portswigger.net/web-security/dom-based/local-file-path-manipulation.
[91] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, "Tranco: A research-oriented top sites ranking hardened against manipulation," in *NDSS*, 2019.
[92] Google Closure Library HTML Sanitizer. https://github.com/google/closure-library/blob/master/closure/goog/html/sanitizer/htmlsanitizer.js.
[93] JS-XSS HTML Sanitizer. https://github.com/leizongmin/js-xss.
[94] Sanitize-HTML Library. https://github.com/apostrophecms/sanitize-html.
[95] Google Caja Sanitizer. https://code.google.com/archive/p/google-caja/wikis/JsHtmlSanitizer.wiki.
[96] Insane HTML Sanitizer. https://github.com/bevacqua/insane.
[97] JavaScript Bleach Sanitizer. https://www.npmjs.com/package/bleach.
[98] Angular-sanitize Library. https://www.npmjs.com/package/bleach.
[99] HTML-Purify Library. https://www.npmjs.com/package/html-purify.
[100] Arcgis HTML Sanitizer. https://www.npmjs.com/package/@esri/arcgis-html-sanitizer.
[101] Python Mozilla Bleach Sanitizer. https://pypi.org/project/bleach/.
[102] LXML Library. https://pypi.org/project/lxml/.
[103] Python HTML-sanitizer Library. https://pypi.org/project/html-sanitizer/.
[104] HTMLLaundry Library. https://pypi.org/project/htmllaundry/.
[105] Django HTML Sanitizer. https://pypi.org/project/django-html_sanitizer/.
[106] PHP HTML Purifier. https://packagist.org/packages/ezyang/htmlpurifier.
[107] PHP HTML-Sanitizer. https://packagist.org/packages/tgalopin/html-sanitizer.
[108] Symfony HTML Sanitizer. https://packagist.org/packages/symfony/html-sanitizer.
[109] HTMLawed Library. https://packagist.org/packages/htmlawed/htmlawed.
[110] Typo3 HTML Sanitizer. https://packagist.org/packages/typo3/html-sanitizer.
[111] HTML Encoder of AntiXSS Library. https://docs.microsoft.com/en-us/dotnet/api/system.web.security.antixss.antixssencoder.htmlencode?view=netframework-4.8.
[112] C# HtmlSanitizer. https://www.nuget.org/packages/HtmlSanitizer.
[113] ASP.NET Ajax Control Toolkit. https://www.nuget.org/packages/AjaxControlToolkit.HtmlEditor.Sanitizer/.
[114] NSoup HTML Parser and Sanitizer for .NET Framework. https://www.nuget.org/packages/NSoup/.
[115] HTMLRuleSanitier Library. https://www.nuget.org/packages/Vereyon.Web.HtmlSanitizer.
[116] JSoup: Java HTML Parser. https://github.com/jhy/jsoup.
[117] OWASP Java HTML Sanitizer. https://github.com/OWASP/java-html-sanitizer.
[118] Java AntiSamy Library. https://github.com/nahsra/antisamy.
[119] HtmlCleaner Library. http://htmlcleaner.sourceforge.net/index.php.
[120] GitHub Octoverse report. https://octoverse.github.com/.
[121] Object Freeze API. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze.
[122] The instanceof Operator. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/instanceof.
[123] The typeof Operator. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof.
[124] The globalThis object. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/globalThis.
[125] The MutationObserver API. https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver.
[126] TheThing. https://github.com/SoheilKhodayari/TheThing.
[127] DOM Clobbering browser testing pipeline. https://github.com/SoheilKhodayari/DOMClobbering.
[128] B. Stock, G. Pellegrino, C. Rossow, M. Johns, and M. Backes, "Hey, you have a problem: On the feasibility of large-scale web vulnerability notification," in *USENIX Security*, 2016, pp. 1015–1032.
[129] F. Li, Z. Durumeric, J. Czyz, M. Karami, M. Bailey, D. McCoy, S. Savage, and V. Paxson, "You've got vulnerability: Exploring effective vulnerability notifications," in *USENIX Security*, 2016.
[130] S. Roth, M. Backes, and B. Stock, "Assessing the impact of script gadgets on csp at scale," in *ACM Asia CCS*, 2020, pp. 420–431.
[131] M. Heiderich, "ToStaticHTML for Everyone! About DOMPurify, Security in the DOM, and Why We Really Need Both," 2016.
[132] Boomerang Library. https://developer.akamai.com/tools/boomerang.
[133] (2018) Client-side CSRF. https://www.facebook.com/notes/facebook-bug-bounty/client-side-csrf/2056804174333798/.

Appendix A.

A.1. Case Studies

This section reports on a few manually vetted case studies of the confirmed attacks. We note that the affected parties have been promptly informed of the vulnerability, and have already patched them (see §7).

GitHub. This vulnerability affects the GitHub Shop and originated when loading the Boomerag JavaScript library [132]. In more details, the code followed the vulnerable pattern G of Table 6, where a variable called `BOOMR` was defined in an inital script that contained a clobberable, invoked native method, and a second script that used the object property `window.BOOMR.url` as the `src` of a dynamically added script. Attackers can escalate this vulnerable pattern to client-side XSS via double clobbering. First, they clobber the invoked native method, causing a runtime error when the browser parses the first script. Therefore, the browser stops parsing the rest of the script and `BOOMR` becomes undefined. Then, attackers can clobber `window.BOOMR.url` and consequently control the script `src` by injecting a DOM Clobbering markup, e.g., `<a id=BOOMR><a id=BOOMR name=url href=malicious.js>`. We discovered that it is possible to inject such non-script markup to the client-side page leveraging the search functionality and the URL query parameters, which were reflected back to the page.

Trello. We discovered that Trello uses a global object property called `window.ClickTaleScriptSource` to programmatically load a script named `wrScript`. However, this property was clobberable as `ClickTaleScriptSource` was an undefined variable following the vulnerable pattern A of Table 6. Finally, we found that it is possible to insert a persistent, non-script markup to overwrite this object property by editing a comment for a card in Trello boards, which resulted in arbitrary client-side code execution.

Fandom. We discovered a DOM Clobbering vulnerability in Fandom affecting the users' message wall that resulted in open redirection. Specifically, the JavaScript program contained an assignment to the `location.href` property of the top-level window, whose value was tainted with a clobberable object property, i.e., `form.elements.targetUsername.value`. Attackers can manipulate the value of this property by, e.g., two nested `iframe` tags that are named `form` and `elements`, and an additional `input` element in the nested frame. The input is named `targetUsername`, and has a `value` containing a malicious URL, which will be set as the window URL. We found that it is possible to inject non-script markup in the page in two distinct ways: (i) attackers can insert persistent payloads using the post functionality in the profile message wall, and (ii) a URL parameter in the path was reflected back to the page without extra validation, enabling transient insertion of clobbering payloads in the page.

A.2. Additional Evaluation Details

Name	</> HTML Tags
TS1	a, abbr, acronym, address, applet, area, article, aside, audio, b, base, basefont, bdi, bdo, bgsound, big, blink, blockquote, br, button, canvas, center, cite, code, command, content, data, datalist, dd, del, details, dfn, dialog, dir, div, dl, dt, element, em, embed, fieldset, figcaption, figure, font, footer, form, h1, header, hgroup, hr, i, iframe, image, img, input, ins, isindex, kbd, keygen, label, legend, li, link, listing, main, map, mark, marquee, menu, menuitem, meta, meter, multicol, nav, nextid, nobr, noembed, noframes, noscript, object, ol, optgroup, option, output, p, param, picture, pre, progress, q, rb, rp, rt, rtc, ruby, s, samp, script, section, select, shadow, slot, small, source, spacer, span, strike, strong, style, sub, summary, sup, table, template, textarea, time, title, track, tt, u, ul, var, video, wbr, xmp
TS2	blockquote, br, button, canvas, center, cite, code, command, content, data, datalist, dd, del, details, dfn, dialog, dir, div, dl, dt, element, em, embed, fieldset, figcaption, figure, font, footer, form, h1, header, hgroup, hr, i, image, img, input, ins, isindex, kbd, keygen, label, legend, li, link, listing, main, map, mark, marquee, menu, menuitem, meta, meter, multicol, nav, nextid, nobr, noembed, noframes, noscript, object, ol, optgroup, option, output, p, param, picture, plaintext, pre, progress, q, rb, rp, rt, rtc, ruby, s, samp, script, section, select, shadow, slot, small, source, spacer, span, strike, strong, style, sub, summary, sup, svg, table, template, textarea, time, title, track, tt, u, ul, var
TS3	button, fieldset, input, output, select, textarea
TS4	image, img, object
TS5	a, abbr, acronym, address, applet, area
TS6	basefont, bgsound, blink
TS7	noembed, noframes, noscript, script, style, template, textarea, title, xmp
TS8	ins, isindex, kbd, keygen, label, legend, li, link, listing, main, map, mark, marquee, menu, menuitem, meta, meter, multicol, nav, nextid, nobr, object, ol, optgroup, option, output, p, param, picture, pre, progress, q, rb, rp, rt, rtc, ruby, s, samp, section, select, shadow, slot, small, source, spacer, span, strike, strong, sub, summary, sup, svg, table, time, track, tt, u, ul, var, video, wbr
TS9	fieldset, figcaption, figure, font, footer, form, h1, header, hgroup, hr, i, image
TS10	form, iframe, image, img, script, style, table, template
TS11	caption, col, colgroup, tbody, td, tfoot, th, thead, tr
TS12	p, param, picture, plaintext, pre, progress, q, rb, rp, rt, rtc, ruby, s, samp, script, section, select, shadow, slot, small, source, spacer, span, strike, strong, style, sub, summary, sup, table, template, textarea, time, title, track, tt, u, ul, var, video, wbr, xmp, a, abbr, acronym, address, applet, area, article, aside, audio, b, base, basefont, bdi, bdo, bgsound, big, blink, blockquote, br, button, canvas, center, cite, code, command, content, data, datalist, dd, del, details, dfn, dialog, dir, div, dl, dt, element, em, embed, fieldset, figcaption, figure, font, footer, form, h1, header, hgroup, hr, i
TS13	h1, header, hgroup, hr, i, image, img, input, ins, isindex, kbd, keygen, label, legend, li, link, listing, main, map, mark, marquee, menu, menuitem, meta, meter, multicol, nav, nextid, nobr, noembed, noframes, noscript, object, ol, optgroup, option, output, p, param, picture, plaintext, pre, progress, q, rb, rp, rt, rtc, ruby, s, samp, script, section, select, shadow, slot, small, source, spacer, span, strike, strong, style, sub, summary, sup, table, template, textarea, time, title, track, tt, u, ul, var, video, wbr, xmp, a, abbr, acronym, address, applet, area, article, aside, audio
TS14	form, b, base, basefont, bdi, bdo, bgsound, big, blink, blockquote, br, button, canvas, center, cite, code, command, content, data, datalist, dd, del, details, dfn, dialog, dir, div, dl, dt, element, em, embed, fieldset, figcaption, figure, font, footer
TS15	data, datalist, dd, del, details, dfn, dialog, dir, div, dl, dt, element, em, embed, fieldset, figcaption, figure, font, footer, form, h1, header, hgroup, hr, i, image, img, input, ins, isindex, kbd, keygen, label, legend, li, link, listing, main, map, mark, marquee, menu, menuitem, meta, meter, multicol, nav, nextid, nobr, noembed, noframes, noscript, object, ol, optgroup, option, output, p, param, picture, plaintext, pre, progress, q, rb, rp, rt, b, base, basefont, bdi, bdo, bgsound, big, blink, blockquote, br, button, canvas, center, cite, code, command, content, rtc, ruby, s, samp, script, section, select, shadow
TS16	rtc, ruby, s, samp, script, section, select, shadow, slot, small, source, spacer, span, strike, strong, style, sub, summary, sup, table, template, textarea, time, title, track, tt, u, ul, var, video, wbr, xmp, a, abbr, acronym, address, applet, area, article, aside, audio, b, base, basefont, bdi, bdo, bgsound, big, blink, blockquote, br, button, canvas, center, cite, code, command, content, data, datalist, dd, del, details, dfn, dialog, dir, div, dl, dt, element, em, embed, fieldset, figcaption, figure, font, footer, form, h1, header, hgroup, hr, i, image
TS17	br, button, canvas, center, cite, code, command, content, data, datalist, dd, del, details, dfn, dialog, dir, div, dl, dt, element, em, embed, fieldset, figcaption, figure, font, footer, form, h1, header, hgroup, hr, i
TS18	address, applet, area, article, aside, audio, b, base, basefont, bdi, bdo, bgsound, big, blink, blockquote, br, button, canvas, center, cite, code, command, content, data, datalist, dd, del, details, dfn, dialog, dir
TS19	form, iframe, script, style, template
TS20	image, img, input, noembed, noframes, noscript

TABLE 7: List of HTML tags used in Table 2 that share the same DOM Clobbering behaviour.

Security Threat	Semantic Type	Description	Reference	JavaScript Sink
Client-side Open Redirect	WIN_LOC_WRITE	Redirecting the Window URL	[36, 37]	`window.location = T`
Websocket Hijacking	WEBSOCK_URL_WRITE	Hijacking Websocket Connections	[80, 81]	`new WebSocket(T)`
Cookie Manipulation	DOC_COOKIE_WRITE	Manipulating Cookie State	[1, 5, 82]	`document.cookie = T`
Doc. Domain Manipulation	DOC_DOMAIN_WRITE	Bypassing SOP	[83, 84]	`document.domain = T`
Client-side JSON Injection	JSON_PARSE	Parsing Untrusted JSON	[85–87]	`JSON.parse(T)`
RegEx Injection	REGEX_BUILD	Injecting Regex for ReDoS	[88, 89]	`new RegExp(T)`
postMessage Manipulation	POST_MSG_WRITE	Manipulating postMessages	[82]	`window.postMessage(T)`
Local File Path Manipulation	FILE_PATH_WRITE	Manipulating Path of Read Files	[90]	`new FileReader().readAsText(T)`
Cross-site Scripting (XSS)	CODE_LOADING	Loading New Scripts	[1, 22, 39]	`script.src = T`

	CODE_EXEC	Executing Arbitrary JavaScript	[1, 7]	`script.textContent = T`
				`eval(T)`
				`setTimeout(T)`
				`setInterval(T)`
				`new Function(T)`

	DOM_NODE_INJECT	Injecting DOM Elements	[1, 7, 12, 87]	`document.write(T)`
				`document.writeln(T)`
				`elm.innerHTML = T`
				`elm.outerHTML = T`
				`elm.insertAdjacentHTML(T)`
				`elm.insertAdjacentElement(T)`
				`elm.replaceChild(T)`
				`elm.append(T)`
				`elm.appendChild(T)`

Web Storage Manipulation	DOC_STORAGE_WRITE	Manipulating Storage State	[1, 5, 82]	`localStorage.setItem()` `sessionStorage.setItem()`

Client-side Request Forgery	REQ	Manipulating Asynchronous Reqs.	[32, 133]	`fetch(T)`
				`XMLHttpRequest.open(T)`
				`asyncRequest(T)`
				`$.ajax(T)`

Legend: T= Tainted Variable;

TABLE 8: Summary of primitive JavaScript sinks and semantic types supported by *TheThing* grouped by the security risk of manipulating the sink object. The list is obtained by aggregating the client-side JavaScript sinks considered in existing literature.

Source	Semantic Type
S1: variable v	CLOB_CUSTOM_VAR
S4, S5: window.v	CLOB_WIN_CUSTOM_VAR
S6: document.v	CLOB_DOC_CUSTOM_VAR
S2: property p	CLOB_NATIVE_PROP
S3: window.p	CLOB_WIN_NATIVE_PROP
S7: document.p	CLOB_DOC_NATIVE_PROP

Legend: S_i= case S_i in Table 3;

TABLE 9: Summary of DOM Clobbering sources and their semantic types based on the seven cases of Table 3.

Sanitizer	Version	Sanitizer	Version
Client-side JS		*Node.js*	
1. DOMPurify	2.3.4	1. Insane	2.6.2
2. Google Closure Lib.	20211201.0.0	2. Bleach	0.3.0
3. JS-XSS	1.0.10	Bower-angular-sanitize	1.8.2
4. Sanitize-HTML	2.6.1	Yahoo html-purify	1.1.0
5. Google Caja	6015	Arcgis	2.9.0
Python		*PHP*	
1. Mozilla Bleach	4.1.0	1. Htmlpurifier	4.14.0
2. LXML	4.7.1	2. Html-sanitizer	1.5.0
3. HTML Sanitizer	1.9.3	3. Symfony HtmlSanitizer	1.0.0
4. Htmllaundry	2.2	4. HTMLawed	1.2
5. Django-html-sanitizer	0.1.5	5. Typo3 Sanitizer	2.0.13
C#		*Java*	
1. AntiXssEncoder	4.3.0	1. Jsoup	1.14.3
2. HtmlSanitizer	7.0.473	2. Java-html-sanitizer	20211018.2
3. AJAX Toolkit	20.1.0	3. Antisamy	1.6.4
4. NSoup	0.8.0	4. HtmlCleaner	2.25
5. HtmlRuleSanitizer	1.6.0.1		

TABLE 10: The specific versions of HTML sanitizers tested in §6

Graph Queries

Task: Identifying Source S_i as in Table 3

Q_{s1} = { n: $n \in$ N $\wedge$ n.type == 'Identifier' $\wedge$ $\exists$ $v \in$ NP $\wedge$ CLOB(v) $\wedge$ n.name == v }

Q_{s2} = { n: $n \in$ N $\wedge$ $\forall$ $s \in$ N, $\nexists$ $e \in$ E, e == edge(s, n) $\wedge$ e.type == 'PDG' $\wedge$ $\exists$ v, $v \notin$ NP $\wedge$ e.value == v $\wedge$ (s.type == 'AssignmentExp' $\vee$ s.type == 'VarDeclaration') }

Q_{s3} = { n: $n \in$ N $\wedge$ n.type == 'MemberExp' $\wedge$ n.object == window $\wedge$ $\exists$ $v \in$ NP $\wedge$ CLOB(v) $\wedge$ n.property == v }

Q_{s4} = { n: $n \in$ N $\wedge$ n.type == 'MemberExp' $\wedge$ n.object == window $\wedge$ $\exists$ $v \notin$ NP $\wedge$ n.property == v $\wedge$ $\forall$ $s \in$ N, $\nexists$ $e \in$ E, e == edge(s, n) $\wedge$ e.type == 'PDG' $\wedge$ e.value == v $\wedge$ (s.type == 'AssignmentExp' $\vee$ s.type == 'VarDeclaration' $\wedge$ s.kind = 'var') }

Q_{s5} = { n: $n \in$ N $\wedge$ n.type == 'MemberExp' $\wedge$ n.object == window $\wedge$ $\exists$ $v \notin$ NP $\wedge$ n.property == v $\wedge$ $\exists$ $s \in$ N, $e \in$ E, e == edge(s, n) $\wedge$ e.type == 'PDG' $\wedge$ e.value == v $\wedge$ (s.type == 'AssignmentExp' $\vee$ s.type == 'VarDeclaration' $\wedge$ s.kind $\in$ {'var', 'let', 'const'})} $\wedge$ $\exists$ $f \in$ N, f.type == 'CallExp' $\wedge$ f.script == s.script $\wedge$ f.name $\in$ NP $\wedge$ CLOB(f.name)

Q_{s6} = { n: $n \in$ N $\wedge$ n.type == 'MemberExp' $\wedge$ n.object == document $\wedge$ $\exists$ $v \notin$ NP $\wedge$ n.property == v }

Q_{s7} = { n: $n \in$ N $\wedge$ n.type == 'MemberExp' $\wedge$ n.object == document $\wedge$ $\exists$ $v \in$ NP $\wedge$ CLOB(v) $\wedge$n.property == v }

Task: Identifying Sink F_i as in Table 8

Q_{sinks} = { n: $n \in$ N $\wedge$ $\exists$ $c \in$ N $\wedge$ hasChild(n, c) $\wedge$ c.type == 'Identifier' $\wedge$ $c \in$ SI }

Task: Identifying Vulnerable Sinks

Q_{vuln} = { n: $n \in$ N $\wedge$ n.type == 'ExpStatement' $\wedge$ $\exists$ $c1$, $c2 \in$ N $\wedge$ hasChild(n, $c1$) $\wedge$ hasChild(n, $c2$) $\wedge$ $c1$.semType == 'SOURCE' $\wedge$ $c2$.semType == 'SINK' }

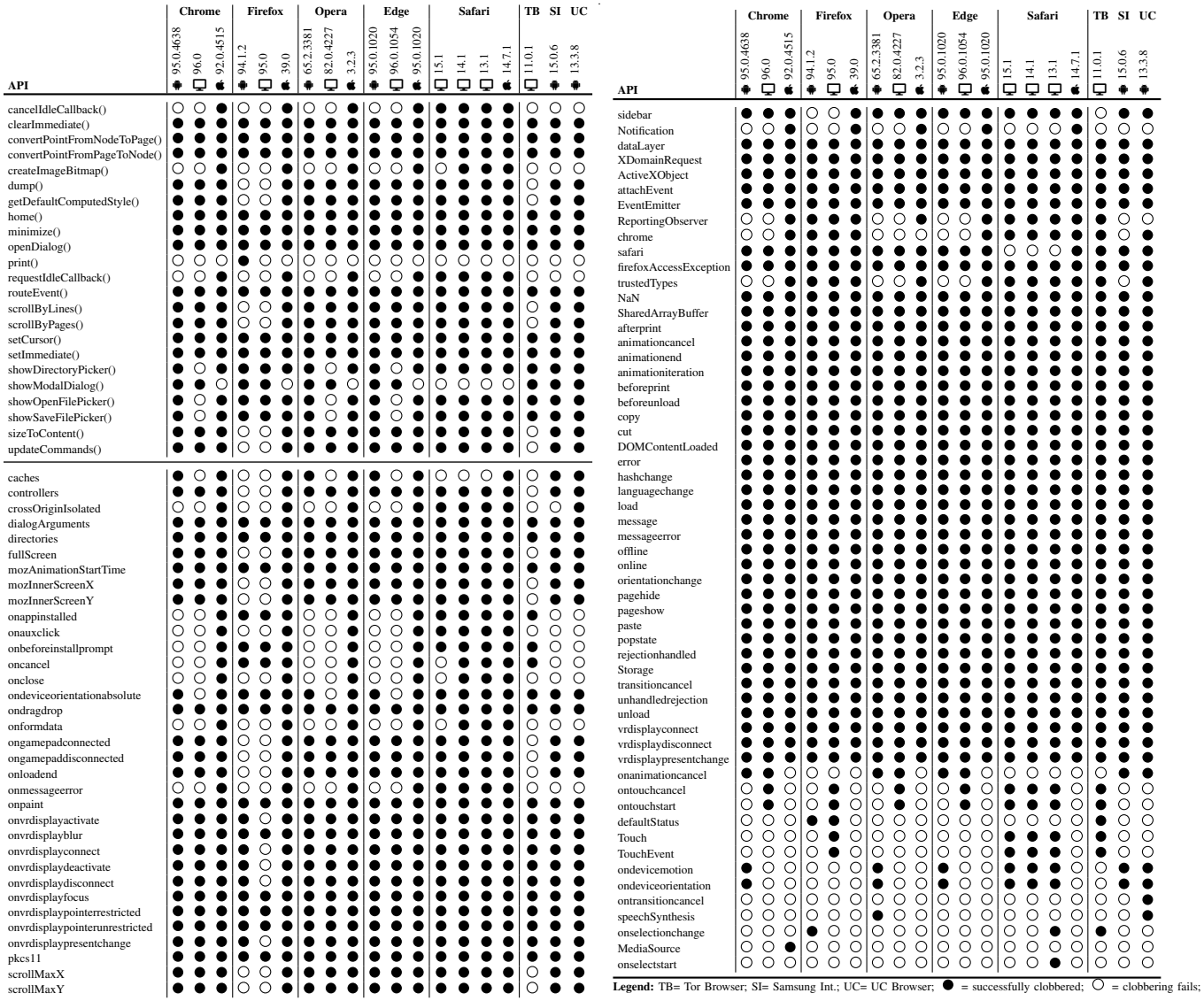
Legend: N, E= HPG nodes, edges; SI= sinks in Table 8; NP= native property; CLOB(v)= v is a clobberable NP according to §4.2.3.

TABLE 11: Excerpt of DOM Clobbering detection queries.

API	Chrome			Firefox			Opera			Edge			Safari				TB	SI	UC	Total
	95.0.4638	96.0	92.0.4515	94.1.2	95.0	39.0	65.2.3381	82.0.4227	3.2.3	95.0.1020	96.0.1054	95.0.1020	15.1	14.1	13.1	14.7.1	11.0.1	15.0.6	13.3.8	
Method	75	72	77	70	69	77	75	72	77	75	72	77	76	77	77	77	69	75	75	79
Property	246	244	256	240	238	255	246	244	254	246	244	255	255	258	260	255	244	246	251	268
Total	321	316	333	310	307	332	321	316	331	321	316	322	331	335	337	332	313	321	326	**347**

Legend: TB= Tor Browser; SI= Samsung Internet; UC= UC Browser;

TABLE 12: Count of clobbered native DOM APIs in mobile and desktop browsers. Browsers with similar behaviours are grouped with the same color.



Legend: TB= Tor Browser; SI= Samsung Int.; UC= UC Browser; ● = successfully clobbered; ○ = clobbering fails;

TABLE 13: List of clobbered `Window` methods and properties in web browsers.