The Great Request Robbery: An Empirical Study of Client-side Request Hijacking Vulnerabilities on the Web

Soheil Khodayari[†], Thomas Barber[*], and Giancarlo Pellegrino[†]
[†]*CISPA Helmholtz Center for Information Security,* [*]*SAP Security Research*
{*soheil.khodayari, pellegrino*}*@cispa.de, thomas.barber@sap.com*

***Abstract*—Request forgery attacks are among the oldest threats to Web applications, traditionally caused by server-side confused deputy vulnerabilities. However, recent advancements in client-side technologies have introduced more subtle variants of request forgery, where attackers exploit input validation flaws in client-side programs to hijack outgoing requests. We have little-to-no information about these client-side variants, their prevalence, impact, and countermeasures, and in this paper we undertake one of the first evaluations of the state of client-side request hijacking on the Web platform.**

Starting with a comprehensive review of browser API capabilities and Web specifications, we systematize request hijacking vulnerabilities and the resulting attacks, identifying 10 distinct vulnerability variants, including seven new ones. Then, we use our systematization to design and implement Sheriff, a static-dynamic tool that detects vulnerable data flows from attacker-controllable inputs to request-sending instructions. We instantiate Sheriff on the top of the Tranco top 10K sites, performing, to our knowledge, the first investigation into the prevalence of request hijacking flaws in the wild. Our study uncovers that request hijacking vulnerabilities are ubiquitous, affecting 9.6% of the top 10K sites. We demonstrate the impact of these vulnerabilities by constructing 67 proof-of-concept exploits across 49 sites, making it possible to mount arbitrary code execution, information leakage, open redirections and CSRF also against popular websites like Microsoft Azure, Starz, Reddit, and Indeed. Finally, we review and evaluate the adoption and efficacy of existing countermeasures against client-side request hijacking attacks, including browser-based solutions like CSP, COOP and COEP, and input validation.

***Index Terms*—CSRF, Request Hijacking, Prevalence, Defenses**

1. Introduction

Request forgery attacks have been one of the most critical threats to web applications since the early days of the Web, where attackers trick victims' browsers into making authenticated, security-sensitive HTTP requests [1–5]. The fundamental vulnerability enabling these attacks is the inability of the server-side component to distinguish unintentional from intentional requests (i.e., the confused deputy flaw [6, 7]), allowing maliciously-forged requests to cause a persistent state change of the web application, such as resetting passwords [8, 9] or deleting data from databases [10, 11]. The recent rapid evolution of client-side technologies has introduced more subtle variants of request forgery vulnerabilities where attackers no longer rely on the confused deputy flaw but instead exploit insufficient input validation vulnerabilities in the client-side JavaScript program to hijack outgoing requests. The research community has only recently started exploring these vulnerabilities, mainly focusing on client-side Cross-Site Request Forgery (CSRF) [12–14] and a corresponding detection and analysis technique [12]. Unfortunately, client-side CSRF is only one instance of the larger issue of request hijacking in web applications, as other types of outgoing HTTP requests exist within JavaScript programs that attackers can hijack, which, to date, remain largely unexplored.

Client-side request hijacking vulnerabilities occur when a JavaScript program uses attacker-controllable inputs, such as URL parameters, to create and send network requests. A closer look at prior work reveals that they primarily focus on *asynchronous* requests generated via the `XMLHttpRequest` [15] and `fetch` [16] APIs, missing other types of outgoing requests and APIs that a JavaScript program can use, e.g., push notifications, web sockets, and server-sent events, including the `sendBeacon` API [17], which accounts for over 35.3% of API calls for asynchronous requests[1]. As a result, we still lack a comprehensive exploration and understanding of this threat on the Web.

Client-side request hijacking attacks are a relatively new phenomenon, with the first documented instance affecting Facebook in 2018 [13], followed by similar incidents involving popular business applications in 2021 [12]. When looking at the countermeasures, prior work has proposed (e.g., [1, 3, 9, 18, 19]) and extensively studied (e.g., [10, 20]) anti-request forgery defenses. However, their focus has been only the traditional request forgery attacks, addressing the confused deputy problem, where the server cannot tell unintentional from intentional requests apart. Unfortunately, the new client-side request hijacks that result from input validation flaws can circumvent these defenses (see, i.e., [12, 14]) and we still miss a systematic and comprehensive exploration of various defense mechanisms against them. Finally, prior

1. We calculated the API usage over Tranco top 10K sites (see §4.3) using the data collection setting detailed in §5.1 and §6.

measurements (i.e., [12]) only focused on a few locally-installed business web applications [21], leaving the in-the-wild impact and prevalence of client-side request hijacking vulnerabilities unclear.

In this paper, we undertake, to the best of our knowledge, the first evaluation of client-side request hijacking vulnerabilities in the wild, covering three main aspects: a systematic exploration of the attack surface, a measurement of vulnerable websites, and a thorough review and evaluation of request hijacking defenses. Starting from a comprehensive survey of browser API capabilities, we systematically examine potential attacks when attackers manipulate one or more inputs of request-sending APIs, covering various types of sensitive requests in modern browsers. Then, we propose Sheriff, a client-side request hijacking detection tool that uses a combination of hybrid program analysis [12] and in-browser dynamic taint tracking [22, 23] for the discovery of potentially-vulnerable data flows and dynamic analysis with API instrumentation [24] for the automated vulnerability verification. We instantiate Sheriff against the Tranco top 10K websites to quantify the prevalence and impact of client-side request hijacking in the wild, processing over 32.4B lines of JavaScript code across 11.5M scripts and 339K webpages. Finally, we identify and evaluate defenses, covering built-in countermeasures offered by browsers and custom defenses implemented by applications at code-level. In particular, we assess the efficacy and adoption of browser policies like Content Security Policy (CSP) [25, 26] and Cross-Origin Opener Policy (COOP) [27], and examine the client-side code to identify insecure input validation practices adopted by developers against request hijacking attacks.

Our results show that the attack surface of client-side request hijacking vulnerabilities is large, with a total of 10 different variants across six request types, of which seven variants are previously unknown, notably hijacking requests of push notifications, window navigations, EventSource, and WebSockets. Furthermore, client-side request hijacking data flows are ubiquitous, affecting 9.6% of the Tranco top 10K websites, with a total of 202K instances across 17.9K webpages. Of these, the new vulnerability types and variants constitute a significant fraction (36.1%), with over 73.3K instances. To demonstrate the significance of these vulnerabilities, we created 67 proof-of-concept exploits in 49 sites, including popular ones like Microsoft Azure, Indeed, Starz, Google DoubleClick, TP-Link, and Reddit, leading to critical consequences such as arbitrary code execution, CSRF, information leakage and open redirections. Finally, the analysis of existing countermeasures suggest that each can only mitigate a fraction of attacks. For example, CSP cannot mitigate over 41% of the information leakage and XSS exploitations of the request hijacking, and COOP and COEP cannot mitigate over 93% and 94.7% of the total request hijacks, respectively. Our results show that developers can fix request hijacking vulnerabilities at code level, and we identify eight insecure input validation patterns to avoid.

Contributions. In summary, this paper makes the following contributions:

Listing 1: Example request hijacking vulnerability in Microsoft Azure.

```
1  var params = (new URL(window.location)).searchParams;
2  var t = params.get("request");
3  if(t != null && t.length){
4      // post message to opener
5      opener && opener.postMessage("reauthPopupOpened", t);
6      // listen for signal
7      window.onmessage = function(){
8          if (event.origin !== opener.origin) return;
9          if (event.data === "sendRequest"){
10             // top-level navigation request
11             document.location.assign(t);}
12     }}
```

- We conduct the first systematic and comprehensive study of client-side request hijacking, covering new vulnerability variants, detection, prevalence, and impact.
- We present Sheriff, an automated detection tool for client-side request hijacking that uncovered 202K vulnerable data flows, affecting 9.6% of the top 10K sites, of which 49 that we manually confirmed to be exploitable, including Microsoft Azure, Indeed, Starz, and Reddit.
- We identify, review and assess the efficacy and adoption of existing countermeasures, showing that CSP and COOP cannot mitigate over 41% and 93% of request hijacking attacks.
- We analyze coding mistakes of the 961 vulnerable websites and extract eight common insecure input validation patterns and practices to avoid.

2. Client-side Request Hijacking

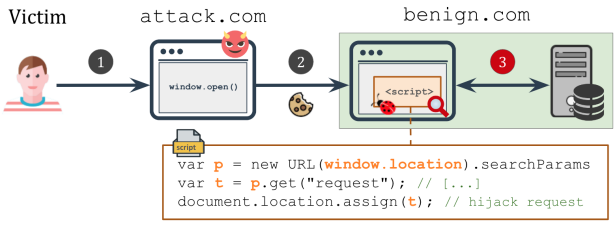
Before presenting our study, we first dissect and introduce the client-side request hijacking vulnerability in §2.1, and then, we present the threat model of this work in §2.2.

2.1. Vulnerability Description

Client-side request hijacking vulnerabilities arise when attackers can trick the client-side JavaScript program into manipulating request-sending APIs with attacker-controlled inputs. The recently proposed client-side CSRF vulnerability [12–14] is a prominent example of such request hijacking, where attackers manipulate `XMLHttpRequest` [15] or `fetch` [16] API parameters, and trigger sensitive actions without user awareness and intention. However, other types of client-side request hijacking also exist.

Listing 1 shows a real example of a request hijacking vulnerability that we discovered in Microsoft Azure (disclosed and patched), where attackers can hijack a top-level HTTP request. In more detail, the code first retrieves a query parameter value from the URL (lines 1-2), and checks that it is not empty (line 3). Then, it sends a postMessage to its `opener` webpage, and waits to receive back the `sendRequest` signal (lines 5-9). Finally, it triggers an HTTP request for navigation by changing the document location to the query parameter value (line 11). The vulnerability originates in the assignment in line 11 because attackers can control the value of query parameters and, ultimately, pick the URL of their choosing for the navigation request. Here, the distinctive characteristic of `location.assign()` as a top-level request

Figure 1: Example request hijacking attack.



introduces additional security risks for cross-site requests, because unlike `XMLHttpRequests` that are constrained by SameSite cookies [10] and Same-Origin Policy [28], top-level requests including `location.assign()` are not, bypassing existing countermeasures.

2.2. Threat Model

In this paper, we consider a *web* attacker [1, 29] who abuses inputs such as URL parameters, window name, document referrer, and postMessages, which is in line with prior work in the area of client-side vulnerabilities and defenses [1, 10, 12, 30–32]. Figure 1 shows an example attack scenario exploiting the vulnerability in Listing 1. First, the attacker prepares a malicious page and lures the victim into visiting it (step 1). The attack page uses the `window.open()` API [33] to open the vulnerable webpage in a new window (step 2), where it injects an attack payload in the query parameter `request` (say attack model A). Alternatively, the attacker can share the malicious URL with victims (instead of using browser APIs) and entice them to click on it, triggering a top-level navigation as shown in [12] (say attack model B). When the page is loaded completely (step 3), the JavaScript code extracts the payload from the query parameter, and triggers a top-level HTTP request towards the payload value, enabling attackers to hijack the original request. Unfortunately, because this request is top-level, browsers will attach cookies to it, circumventing the SameSite policy [10, 34]. In particular, in attack model A, the `SameSite=Lax` policy (default in Chromium-based browsers) attaches cookies to `window.open()` requests but `SameSite=Strict` policy can mitigate that. However, in scenario B, even `SameSite=Strict` is not sufficient, as cookies are always attached to same-site requests. Consequently, the attacker obtains CSRF by sending arbitrary requests to any security-sensitive endpoint, resulting in compromise of database integrity (e.g., deleting VMs, and changing user settings in Azure). Note that cross-origin policies like CORS (i.e., `access-control-*` headers [35]) allow a server to restrict access of any origin other than its own, thus are ineffective against CSRF exploitations that abuse the same-origin requests.

Then, in this paper, we consider other types of URLs that are not based on the `http` scheme. For example, the `location.assign()` API also accepts URLs with the `javascript` scheme, which enables attackers to escalate request API hijacking to arbitrary client-side code execution if there is no or improper input validation, e.g., by injecting `javascript:alert(document.cookie)` in the query parameter `request` in Listing 1. Accordingly, as this example highlights, hijacking a request API can have a wide range of consequences, including cross-site request forgery, client-side code execution, open redirection, and sensitive information leakage–to name only a few examples. As we will show in §7, these types of request hijacking attacks could be mitigated by constraining request APIs with *opt-in* security policies, e.g., using the CSP `connect-src` directive [36].

3. Problem Statement

This paper answers the following research questions:

RQ1: Browser Capabilities and Attack Systematization. The recently proposed client-side CSRF vulnerability [12–14] allows attackers to generate arbitrary HTTP requests by manipulating JavaScript program input parameters. However, client-side CSRF is just one instance of the broader issue of request hijacking in client-side code, i.e., JavaScript programs can perform different types of requests (e.g., asynchronous vs top-level requests or socket connections) using numerous APIs (e.g., `XMLHttpRequest` vs `sendBeacon`), which presents a diverse threat landscape. In this paper, we take a step back and study client-side request hijacking vulnerabilities. First, we look at various browser methods and APIs for sending requests, and label each with specific capabilities (e.g., accept `javascript` URIs, allow setting the request body, etc). Then, we review existing literature and conduct a comprehensive threat modeling analysis, systematically assessing the security risks that emerge when an attacker can manipulate various fields of request-sending APIs.

RQ2: Detection, Prevalence, and Impact. Despite being aware of client-side CSRF since 2018 [13], we still lack a clear understanding of its prevalence and severity across the Web on a large-scale. Unsurprisingly and by extension, we have little-to-no information about the overall impact and pervasiveness of the broader issue of request hijacking in real websites. In this paper, we aim to fill this gap by quantifying the prevalence of request hijacking in the wild, identifying vulnerable behaviours, and investigating their impact to gain insights into the underlying issues and factors that affect the security posture of web applications.

RQ3: Defenses and Effectiveness. While numerous research efforts studied request forgery countermeasures (e.g., [1, 3, 9, 10, 18–20]), their focus has been only the traditional request forgery attacks that abuse the confused deputy flaw, and hence, we still lack a comprehensive understanding of the protective coverage of various defenses mechanisms against client-side variants of the request hijacks. As the final part of our paper, we systematically assess existing defenses and their efficacy leveraging data collected from the previous answers. In particular, we measure the efficacy and adoption of browser-based policies, such as CSP [25], COOP [27] and COEP [37], and examine the discovered vulnerabilities to uncover insecure input validation patterns and practices adopted by developers.

4. API Capabilities and Attack Systematization

In this section we address RQ1, where we systematically assess modern web browser APIs and their capabilities for sending various types of client-side requests (§4.1). Then, we examine each API call to evaluate the resulting vulnerabilities and attacks when an attacker controls one or more API inputs (§4.2). Finally, we assess the prevalence of request API usage on the Web platform (§4.3).

4.1. Browser API Capabilities

Client-side web applications have access to a wide range of browser functionality via JavaScript Web APIs. An important group of these APIs is responsible for creating and sending network requests, which malicious actors could exploit for request hijacking. To compile a comprehensive list of request-sending APIs susceptible to such abuse, we performed a systematic search of the Web request specifications [16, 17, 38–41] from WHATWG [42] and W3C [43] repositories. Our search focused on identifying JavaScript Web APIs capable of creating network requests.

As a result, we identified a total of 10 request APIs across six broad request categories. Each API features different characteristics in terms of their supported capabilities, e.g., the network schemes and methods available for a given API, the configurable fields of the request (e.g., body and headers), and the constraints the APIs may be subject to *by default*, such as the Same-Origin Policy [28]. In this section, we focus on *default* constraints the request APIs are subject to. The goal is to identify possible attacks under default settings. In §7, we explain the role other *opt-in* security mechanisms like CSP, COOP, and COEP could play to mitigate the request hijacking attacks. We note that these policies are opt-in mechanisms that can influence the exploitation of the vulnerability, not the presence of the underlying software weakness. For this reason, we do not consider them at the time of the threat analysis and vulnerability detection. The resulting APIs are summarized in Table 1. By examining these request APIs, we uncover potential entry points for various forms of request hijacking, and their consequences.

4.2. Systematization of Request Hijacking Attacks

In this section we examine the security impact assuming the threat model presented in §2.2. That is, an attacker can control the URL (and if applicable, the body and header) of network requests for each of the APIs discovered in §4.1. First, we systematically surveyed the existing academic [1, 2, 10, 12, 26, 30, 31, 44, 46, 47, 51–53] and non-academic [45, 48–50, 55] literature, looking for known attacks leveraging these APIs. Then, we conducted an in-depth analysis of the threat landscape, where we examined the potential attacks resulting from an attacker's capability to manipulate different fields of each request-sending API.

As a result, we identify a total of 10 client-side request hijacking vulnerability variants, of which only three are previously known. Table 2 presents the list of request hijacking vulnerabilities, together with the responsible APIs and attacks which are made possible as a consequence of each vulnerability explained in more detail in the following. We refer interested readers to §A.2 for examples of attacks.

4.2.1. Asynchronous Requests. Asynchronous requests such as `XMLHttpRequest` [39] or the low level `fetch` API [16] are typically used to communicate with web services such as REST APIs, without causing the top-level page to reload. Attacker manipulation of the URL, body, or header of asynchronous requests in client-side JavaScript programs can lead to the victim performing unwanted actions on behalf of the attacker, i.e., client-side CSRF [12, 13], similarly to their traditional counterpart [1, 2, 20, 56]

We are currently unaware of studies exploring the manipulation of `sendBeacon` API [17] for client-side CSRF attacks. Furthermore, attacker control of asynchronous request URLs can also lead to information leakage, which was not considered by prior works (e.g., [12]). In this case the attacker manipulates the URL host to point to a malicious server, where they can access sensitive information stored in the request header or body, e.g., login credentials, personally identifiable information (PIIs), and CSRF tokens.

4.2.2. Push Requests. Push notifications [40] allow a web server to asynchronously send messages to a browser, even if the web application is not currently loaded. The Push API requires subscription to a push service via an HTTP POST request, and a browser can request new messages via HTTP GET requests. If Push subscriptions do not have anti-request forgery tokens, attackers can conduct classical CSRF attacks [45].

While not explored before, similar attacks are possible in the context of client-side programs. For example, when creating a Push subscription, the client sends information such as the subscription endpoint and public key in the body of an HTTP POST request. Attacker control of the request body would allow manipulation of these parameters and hence CSRF [45–47], e.g., by overwriting the subscription endpoint the application saves in the backend, the attacker can redirect Push messages to an arbitrary server. We note that in the case of Push requests, the URL must take a specific value (i.e., the endpoint listening for Push subscriptions), so URL manipulation will not usually lead to CSRF attacks. However, setting an invalid value enables attackers to cause a persistent client-side DoS, which can be mitigated when users reset their browser notification permissions. In addition, control of the Push URL could lead to information leakage if the request is redirected to an attacker-controlled server, e.g., the leaked Push endpoint and encryption key can be exploited to send malicious messages to the victim's browser.

4.2.3. Server-Sent Events. Server-sent events (SSEs) [38, 48] allow servers to push messages to a browser at any time, without waiting for a new request. SSEs are initiated via an HTTP GET request to a URL specified in the `EventSource` constructor. By manipulating the EventSource URL, attackers can redirect the request to

	API	Req. Type	Capabilities: Schemes	Methods	URL	Body	Header	SSC	SOP	Specs	# Sites	# Pages	# Calls
#1	Location Href	Top-Level Navigation	HTTP(S), JS	GET	●	○	○	○	●	[38] §7.2.4	8,044	214,554	1,096,306
#2	XMLHttpRequest	Async. Request	HTTP(S)	Any	●	●	●	●	●	[39] §3.5	7,522	407,819	2,884,556
#3	sendBeacon	Async. Request	HTTP(S)	POST	●	●	○	●	●	[17] §3.1	7,061	291,580	2,824,388
#4	Window Open	Window Navigation	HTTP(S)	GET	●	○	○	○	●	[38] §7.2.2.1	6,972	162,153	559,592
#5	Fetch	Async. Request	HTTP(S)	Any	●	●	●	●	●	[16] §5.4	5,215	105,463	403,701
#6	Push	Push Subscription	HTTP(S)	GET, POST	●	●	○	●	●	[40] §3.3	1,528	23,566	40,567
#7	WebSocket	Socket Connection	WS(S)	GET	●	●	○	○	○	[41] §3.1	1,280	33,724	145,713
#8	Location Assign	Top-Level Navigation	HTTP(S), JS	GET	●	○	○	○	●	[38] §7.2.4	987	10,092	22,309
#9	Location Replace	Top-Level Navigation	HTTP(S), JS	GET	●	○	○	○	●	[38] §7.2.4	731	6,421	14,309
#10	EventSource	Server-Sent Event	HTTP(S)	GET	●	○	○	●	●	[38] §9.2	453	1,690	5,503

Legend: SSC= SameSite Cookies; SOP= Same-Origin Policy; ● = Supported Capability or Applicable Constraint; ○ = Otherwise.

TABLE 1: Overview of security-sensitive JavaScript APIs that inititate client-side requests, along with their supported capabilities, default constraints and usage in top 10K Tranco websites (Cf. §6). The table is ordered by the API usage in the wild.

Vulnerability	Reqs.	CSRF	XSS	WS Hijack	SSE Hijack	Inf. Leak	Open Red.	DoS	Related Ref.
⊞ Forge. Async Req. URL	#2, 3, 5	●	○	○	○	●	○	○	[10, 12, 44]
⊞ Forge. Async Req. Body	#2, 3, 5	●	○	○	○	○	○	○	[1, 2, 12, 44]
✚ Forge. Async Req. Header	#2, 5	●	○	○	○	○	○	○	-
✚ Forge. Push Req. URL	#6	○	○	○	○	●	○	●	-
✚ Forge. Push Req. Body	#6	●	○	○	○	○	○	○	[45–47]
✚ Forge. EventSource URL	#10	○	○	○	●	●	○	○	[48]
✚ Forge. WebSocket URL	#7	○	○	●	○	●	○	○	-
✚ Forge. WebSocket Body	#7	●	○	●	○	○	○	○	[44, 49–52]
Forge. Location URL	#1, 8, 9	●	●	○	○	○	●	○	[30, 53, 54]
✚ Forge. Window Open URL	#4	●	●	○	○	○	●	○	-

Legend: Forge.= Forgeable; SSE= Server-Sent Event; WS= WebSocket; #i= row i in Table 1; ● = Applicable Attack; ○ = Otherwise.

TABLE 2: Overview of client-side request hijacking vulnerabilities and attacks. Rows marked with ✚ are new (i.e., client-side variants of) vulnerabilities and ⊞ represent vulnerabilities for which we consider a new API or exploitation. For new vulnerabilities, related references refer to their server-side vulnerability counterparts.

a malicious server and achieve SSE hijacking, whereby malicious events can be sent to the victim's browser. Similarly to asynchronous and push requests, redirection of the EventSource URL can also lead to information leakage.

4.2.4. Web Sockets. WebSockets [41, 57] enable full-duplex, event-driven communications between browsers and servers, initiated via an HTTP GET request. An attacker controlling the WebSocket connection can perform cross-site WebSocket hijacks (CSWSH) [44, 49, 52]. In this scenario, an attacker can embed a WebSocket to a target website on their own site. When a victim visits this malicious site, the victim's browser is tricked into performing authenticated actions on behalf of the attacker. In contrast to write-only CSRF attacks, CSWSH allows full read/write communication. In the context of client-side vulnerabilities, we show that similar attacks are possible if the attacker can control the URL used to perform the initial handshake, redirecting the request to a malicious server. As the attacker controls the WebSocket, this can also lead to information leakage from the victim's browser. Finally, controlling the data used in WebSocket messages leads to message hijacking and potentially CSRF.

4.2.5. Top-Level Navigation Requests. Top-level navigation requests via the `location` API allow manipulation of the current browser URL, and trigger a new HTTP GET request when called [38]. Attacks leveraging this category of requests have been considered in the past. For example, manipulating the `location` URL can lead to client-side XSS by exploiting the `javascript` protocol if the entire URL is controlled by the attacker [30, 53]. Alternatively, replacing the full URL with a malicious URL will force an open redirect of the browser to a different site. Finally, even partial hijack of the URL (e.g., query parameters) can trigger the application to perform actions on behalf of the attacker leading to CSRF. This usually occurs if the application implements state-changing GET requests [56] or allows forging POST requests with GET where it incorrectly accepts and processes incoming requests regardless of their HTTP method, as shown in [10].

4.2.6. Window Navigation Requests. The `window.open` API triggers a top level HTTP request in a specific browser context, such as a new window or the current one (i.e., redirection). Similarly to the `location` API, control of the `window.open` API could also lead to CSRF, XSS and open redirects. However, unlike `location`, we are not aware of previous work that has studied this vulnerability.

4.3. Request API Prevalence

Having examined the web APIs which are susceptible to request hijacking, we also measured their prevalence in the wild. The last three columns of Table 1 list the number of sites, pages and calls of a particular API found in the top 10K Tranco websites. More details on the dataset and crawling strategy can be found in §6.

Overall, we find 9,901 domains which contain at least one API related to client-side requests, with a total of ∼7.9M API calls across 1,032,795≈1M webpages. Top-level navigation requests via `location.href` are the most widespread, being present on more than 8K sites. Asynchronous requests via the `XMLHttpRequest` API are the most widely-used, with almost 3M calls across over 400K pages. We observed

that request hijacking threats have not been considered for over 44.7% of API calls by prior work given the new vulnerability variants presented in Table 2. The widespread usage of request-related APIs in the wild, coupled with a wide variety of potential vulnerabilities, presents a tantalizing attack surface for hackers. The remainder of this paper is dedicated to techniques for the detection and evaluation of these vulnerabilities in the wild.

5. Vulnerability Detection

Starting from our systematization of vulnerabilities presented in §4, we now formulate our approach to detect and study request hijacking vulnerabilities, thereby addressing the first part of RQ2. Client-side request hijacking vulnerabilities arise due to the presence of insecure data flows from attacker-controlled inputs to request-sending instructions. In this paper, we design and implement an open-source, static-dynamic analysis tool, called Sheriff, to detect such insecure data flows.

Figure 2 depicts the architecture of Sheriff. Broadly, it comprises four main components: (1) a data collection module that gathers Web resources and dynamic taint flows from webpages, (2) a data modeling module that processes the collected data to identify and model unique webpages, creating a property graph for each one, (3) a vulnerability analysis module that traverses this graph following the propagation of unvalidated data flows from input sources to request-sending functions, and finally (4) a dynamic verification module that confirms the potential forgeability of requests. The rest of this section describes each component.

5.1. Data Collection

The first step of our analysis pipeline involves collecting client-side code and runtime values (e.g., DOM snapshots) of web applications for security testing. Starting from a list of sites under test like Tranco [58], Sheriff instantiates N crawling workers and continues orchestration until all input sites have been crawled. We developed a taint-aware crawler based on Playwright [59], an instrumented version of Firefox known as Foxhound (v98.0.2) [22, 23], and Firefox DevTools [60]. Since Foxhound does not provide instrumentation support for all request APIs listed in Table 1, we added further instrumentation to provide taint tracking support for these APIs (hereafter, Foxhound$^+$). Given a domain as input, the crawler visits webpages with a depth-first strategy, and stops when it does not find new URLs or visited a maximum of 200 URLs per site. During the visit, the crawler collects the following information: webpage resources (e.g., scripts), DOM snapshots, global objects' properties, event traces, network requests and responses, and finally dynamic taint flows from program inputs to security-sensitive instructions, such as request-sending functions.

Our crawler does not create accounts or login since manual creation and maintenance of sign-up and sign-in scripts is brittle and challenging, particularly when dealing with thousands of applications. This limitation is in line with the state-of-the-art of security testing at scale (e.g., [23, 31, 32]).

5.2. Data Modeling

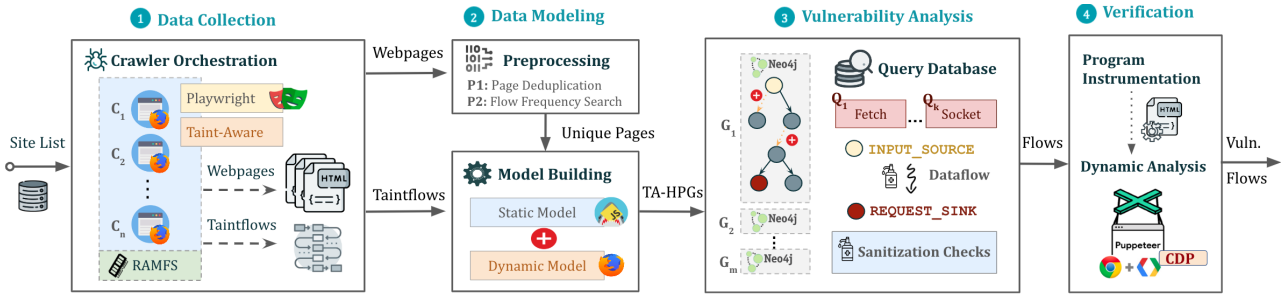
5.2.1. Preprocessing. Given the webpages' data collected by the crawler, Sheriff performs data pre-processing for efficiency and scalability reasons. For example, Sheriff pre-processes the client-side code to filter out near-duplicate webpages [61, 62] by comparing SHA-256 script hashes, which allows it to focus on pages with distinct JavaScript code, reducing the overall effort for program analysis. Similarly, Sheriff can perform other types of data preprocessing, such as (custom) search-based filtering of data or code normalization as used in prior work [12].

5.2.2. Model Building. After removing duplicate webpages, Sheriff creates a property graph [2, 63, 64] of the client-side program under test, capturing both static and dynamic program behaviours, known as Hybrid Property Graph (HPG) [12]. Sheriff instantiates a pool of workers to generate HPGs, using an extended engine of JAW [12]. Then, these HPGs are enriched with taint flow information provided by Foxhound$^+$ to patch missing HPG edges due to static analysis shortcomings, and finally stored in a Neo4j [65] graph database which we can query for security testing.

Hybrid Property Graphs. HPGs [12] are unified representations of client-side JavaScript programs that integrate static code representations and runtime state values into a graph-based structure. State values are concrete program values observed during execution, e.g., web storage values and cookies. HPGs integrate several static code representations, i.e., Abstract Syntax Tree (AST), Control Flow Graph (CFG), Call Graph (CG), Program Dependence Graph (PDG), and Event Registration, Dispatch and Dependency Graph (ERDDG), that collectively capture the program's syntactical nesting, execution order, function call relationships, data flow and control dependencies, and event-driven control transfer, respectively. HPGs also incorporate Semantic Types, which are labels assigned to nodes (e.g., sinks and sources) to capture the semantic meaning of instructions. Encoded as a directed graph, HPGs employ a labeled property graph structure, where nodes and edges possess labels and key-value properties [63, 66].

Taintflow-Augmented (TA) HPGs. In this paper, we formulate the request hijacking vulnerability detection task over HPGs, where we intend to identify request-sending instructions that are triggered at page load, and that are susceptible to manipulation by attackers through program inputs. Unfortunately, conducting such inter-procedural reachability and data flow analysis tasks is non-trivial due to the dynamic nature of client-side JavaScript programs (e.g., [12, 67–71]). While HPG state values (i.e., environment properties and event traces [12]) help to alleviate many of JavaScript static analysis shortcomings (e.g., imprecise control and data flow dependencies) by reasoning on concrete object snapshots (e.g., points-to analysis and triggered event handlers), they are

Figure 2: Architecture of Sheriff.



not sufficient to identify many of the other missing call and data flow connections in the graph. Accordingly, in this paper, we use fully-fledged, in-browser dynamic taint tracking to further augment HPGs by adding supplementary edges and labels to nodes (e.g., to mark reachability, semantic types and runtime variable values), thus reconstructing missing connections that are reachable at page load, which are otherwise missed by static analysis.

TA-HPG Construction. We used Foxhound$^+$ to collect dynamic taint flows from input sources to all sink types, including those that are not request-sending instructions (Cf. Table 11), so that we can complement as many potentially missing elements as possible in the HPG. Specifically, we first extract the dynamic call graph and data flow graph from Foxhound$^+$, and merge them with static call graph and PDG, respectively. To do the match between the dynamic and static graphs for merging, we first determine in which script file an instruction or node is located by comparing the script hash in the two models and, then, use the line of code to determine the top-level (i.e., CFG) node in the HPG for that instruction. When merging the dynamic data flow graph with PDG, we create data dependency edges if an edge is missing, with the labels being the variable name reported in the taint flow whose data is propagated. Conversely, if a PDG edge already exists between two nodes, we add a label to that edge marking the runtime value of the propagated variable.

Similarly, when merging the two call graphs, we create a new edge if it is missing and label it with the invoked function and parameter names as well as concrete parameter values of the function call. However, when the call edge exists in the HPG, we only enrich its information by adding the runtime values of the call site parameters. Finally, we added labels to all sources and sink nodes as semantic types, capturing the semantic of those instructions, e.g., the type `RD_DOC_URL` is set for instructions that read the value of `document.URI`, and then propagated to other HPG nodes following the calculation of the program. We refer interested readers to §A.1 for more details on TA-HPGs. As we will show in §6, this configuration facilitates a comprehensive representation of program dynamics, enabling enhanced analysis capabilities for vulnerability discovery.

Static Analysis Engine Enhancements. To enhance JAW's HPG generation, we made several modifications addressing incomplete ES6 support for improved control transfer modeling and data flow analysis. For example, we added support for asynchronous operations using the `Promise` object and `setTimeout()` callbacks [72], improving the precision of call graph and PDG edges. Additionally, we applied multiple optimizations to improve scalability, such as handling inefficiencies in iteration constructs during the PDG construction and managing Neo4j graph databases in parallel by creating an orchestrator using ineo [73]. Overall, these modifications addressed several of the shortcomings of JAW, enabling more precise analysis and improved scalability in the construction of HPGs.

5.3. Vulnerability Analysis

After modeling the client-side code as a TA-HPG, we define the task of detecting request hijacking vulnerabilities as a graph traversal problem. Specifically, we intend to search for program instructions that send sensitive requests at page load, whose parameters originate from attacker-controlled program inputs. As the first step, we identify TA-HPG sources that read attacker-controlled inputs (Cf. §2.2), and assign them a relevant semantic type similarly to JAW [12], e.g., we set a label named `RD_WIN_LOC` for instructions that read the URL through `window.location` API. Then, given a list of browser APIs that are used for sending requests (Cf. Table 1), Sheriff searches the TA-HPG to identify nodes using these APIs, and marks them as a sink by assigning them a relevant semantic type, e.g., the label `WR_ASYNC_REQ_URL` is set for instructions that write the URL of an asynchronous request, such as `XMLHttpRequest.open()`.

Finally, to discover vulnerable paths, Sheriff performs data flow analysis by propagating semantic types from sources to sinks over PDG, CFG, CG, and ERDDG edges, and selects unvalidated paths where a node with a sink semantic type is tainted with a source type and picks up the attacker-controlled values. Then, Sheriff performs reachability analysis to check if the vulnerable path may correspond to lines of code executed at page load. To do that, it starts from both source and sink nodes and follows backward CFG, ERDDG, and CG edges until it reaches the CFG entry node or there are no longer edges matching the criteria to backtrack, and selects data flows where both the source and sink are reachable nodes. Ultimately, this component outputs a set of paths with potential data flows from a source to a request sink.

5.4. Vulnerability Verification

Given a set of candidate request hijacking data flows, the goal of this step is to confirm the feasibility of each flow dynamically and eliminate potential false positives. We relied on Playwright [59] and Chrome DevTools Protocol [24] to perform runtime monitoring, where we instrumented browser APIs responsible for sending requests (Cf. Table 1) and intercepted network messages that occur at page load. To minimize risks and ensure responsible conduct, the verification module instruments request APIs to only log request parameters, without actually sending any requests to server-side. This approach mitigates ethical concerns by preventing unintended interactions. Specifically, for each webpage, we first compare the script hashes in our dataset and the live webpage. If they match, we perform runtime monitoring against the live version. Otherwise, we test the local snapshot. This approach ensures that the live webpage remained unchanged since our data collection, mitigating the time-of-check to time-of-use issue.

Then, for each request hijacking data flow, we input a benign token to the corresponding source, load the webpage in an instrumented browser controlled with Playwright and search for the token in the client-side request to check whether the manipulated inputs are observed. We test each candidate data flow both in the affected webpage and all its near-duplicate pages, which have the same set of scripts but potentially different DOM environments (Cf. §5.2.1). By doing so, we switch DOM trees when testing the data flow within the affected JavaScript program, as the DOM environment can affect the execution of the program. To enable this approach, we need to provide the input differently depending on the type of the source, i.e., URL parameters, postMessages, document referrer and window name (Cf. §2.2). For example, in case of URL parameters, we can control them directly, and load the manipulated URL for testing. For other sources, we load a test webpage in the browser, which uses `window.open()` [33] to open the target webpage in a new window and set the window name through `window.name` API [74] or send postMessages to the opened window [32]. Alternatively, the test page can redirect to the target webpage and control the document referrer leveraging the URL of the test page. Finally, we perform manual analysis to validate the decision reported by Sheriff and examine the exploitability of the reported data flows.

6. Empirical Evaluation

This section addresses the second part of RQ2 (Cf. §3), where we conduct the largest-to-date study to quantify the prevalence and impact of request hijacking vulnerabilities in the wild. To accomplish this, we utilized the Tranco site list downloaded on Sept. 29, 2022 (ID: N7QWW) [58], where we first selected the top 10K domains by excluding duplicate versions of websites (e.g., *google.com* and *google.co.uk*), and then instantiated Sheriff for each of them. We started our crawling infrastructure of §5.1 in Oct. 2022 by deploying 100 parallel browser instances. To ensure comprehensive coverage, we made up to three repeated attempts for each failed crawling website, followed by a detailed manual analysis. The entire data collection process spanned approximately six weeks. The rest of this section details our findings.

Top 10K Sites	Raw Data	Dedupl.	Top 50 Flows
# Webpages	1,034,521	867,455	339,267
# Scripts	46.1 M	36.7 M	11.5 M
# LoC	129.8 B	104.1 B	32.4 B
# Taint Flows	43,143,773	35,209,216	21,673,167
# Req. Flows	8,024,030	7,205,914	3,318,747

Legend: Dedupl.= Page Deduplication.

TABLE 3: Summary of the collected data and preprocessing steps.

6.1. Data Collection and Processing

Table 3 summarizes the results of data collection and modeling steps. Starting with the 10K seed URLs, Sheriff obtained a grand total of 1,034,521≈1M webpages across all websites. The number of pages per site spanned from 1 to 200, averaging at 103 pages. These 1M pages contained around 46.1M scripts with over 129.8B LoC. Page de-duplication (Cf. §5.2.1) enabled us to focus on pages with unique sets of scripts and reduced the size of the dataset by about 17%, that is, out of the total 1M webpages, 867,455 pages were unique.

Considering the extensive size of the raw data and the need to analyze hundreds of thousands of webpages, we further reduced the size of our testbed by focusing our testing efforts on the top 50 pages of each site that exhibit the greatest frequency of dynamic taint flows (originating from input sources and reaching request-sending sinks), which is based on the higher probability of these pages containing vulnerabilities. In summary, the 867K unique pages contained ∼7.2M dynamic taint flows to request-sending sinks which we used for our page selection. Accordingly, the 867K webpages were filtered to 339,267 pages. Out of these, Sheriff extracted 11,544,754 scripts (32.4B LoC) and 21.6M dynamic taint flows, that we can use to enrich HPGs in order to remediate missing connections that are not discovered by static analysis. Out of these 21.6M taint flows, 3,318,747 flows contain request-sending sinks, which can indicate the presence of request hijacking vulnerabilities. In total, Sheriff processed an average of 34 scripts and 95K LoC per page, generating 339,267 TA-HPGs.

6.2. Prevalence in the Wild

After TA-HPG construction, Sheriff performed graph traversals for vulnerability discovery following §5.3. In summary, Sheriff identified an average of 23 request-sending sinks and 65 sources per webpage, totaling about 7.9M sinks and 22.3M sources. Among these, static analysis found a total of 236,427 potential data flows from sources to sinks, of which ∼86% (i.e., 202,834) were verified following runtime experiments. These vulnerable data flows affected around

			Flows						Breakdown				
	Vulnerability	Sinks	WURL	WN	DR	PM	Total	Verified	Dynamic	Mixed	Static	Pages	Sites
⊞	Forge. Async Req. URL	6,112,645	106,218	105	1,232	46	107,601	91,688	47,631	8,037	36,020	12,908	616
⊞	Forge. Async Req. Body	4,584,483	76,517	428	5,209	6,564	88,718	78,240	63,308	7,442	7,490	9,510	819
✚	Forge. Window Open URL	559,592	20,574	21	76	3	20,674	16,566	49	652	15,865	8,846	365
	Forge. Location URL	231,067	4,533	300	108	8	4,949	4,079	1,157	131	2,791	2,610	324
✚	Forge. Async Req. Header	119,855	2,401	5	42	372	2,820	2,446	1,710	135	601	1,587	107
✚	Forge. WebSocket URL	145,713	5,322	32	320	807	6,482	5,520	2,865	543	2,113	1,096	56
✚	Forge. WebSocket Body	145,713	2,867	18	172	434	3,490	2,973	1,543	292	1,137	590	30
✚	Forge. Push Req. URL	40,567	592	93	0	0	685	539	0	530	9	497	25
✚	Forge. Push Req. Body	26,441	94	61	2	0	157	119	0	115	4	101	9
✚	Forge. EventSource URL	5,503	680	2	36	133	851	664	387	66	211	56	3
	Total	7,996,944	219,798	1,065	7,197	8,367	236,427	**202,834**	118,650	17,943	66,241	**17,805**	**961**

Legend: Forge.= Forgeable; WURL= Window URL; WN= Window Name; DR= Document Referrer; PM= postMessage.

TABLE 4: Summary of client-side request hijacking data flows in top 10K sites. The table shows the total number of data flows from input sources (columns 3-6) to request sinks (Cf. Table 11) as well as the affected webpages and sites. Rows marked with ✚ and ⊞ represent new vulnerabilitiy types proposed by our work and variants for which we also consider a new API, respectively. The table also shows the distribution of data flow paths in the TA-HPG (static, dynamic, or mixed) based on the type of edges involved in the flow, highlighting the contribution of dynamic information for vulnerability discovery.

5.2% of the tested webpages (17,805 out of 339,267) and 9.6% of the sites (961 out of 10K), which is alarming. Table 4 presents a summary of the results.

Types of Hijacked Requests. Among the various types of requests that can be hijacked, asynchronous requests are the most widespread (85%), with over 172K instances across 905 sites. Interestingly, forged window loads are the second-most prevalent (8.2%), accounting for 16.5K flows in 365 sites. At the other extreme, hijacked push requests and EventSource occur the least often, each affecting only about 0.3% of the flows across 25 and three sites, respectively. Finally, hijacked web sockets and top-level requests demonstrated a moderate level of prevalence, corresponding to about 6% of the vulnerable data flows in total.

New Vulnerability Types. We observed that the new vulnerability types and variants listed in Table 2 constitute a significant fraction (i.e., 36.1%) of the request hijacks. First, the new vulnerability types account for over 14.2% (35,159) of the total 236K discovered cases. Among these, Sheriff verified 28,827 vulnerable data flows across 10,925 webpages and 439 sites, highlighting the widespread occurrence of the new vulnerabilities. Then, 21.9% of the request hijacks are new variants where we considered a new browser API.

Vulnerability Impact. We found that the 202K vulnerable data flows can have different security implications (Cf. Table 2), where each vulnerability could lead to multiple consequences through different exploitations, amplifying the potential risks. The most common consequence is client-side CSRF, for which 96% of the vulnerabilities (i.e. 196K) can be abused. However, 48.5% of the hijackable requests can be abused for information leakage too, as the attacker can control the endpoint to which the request is sent to, and consequently steal the sensitive information contained in the request body, such as CSRF tokens, PII, push endpoint and encryption key, as we will show in §6.4. In comparison, the least common consequence is persistent DoS on push subscriptions that accounts for 0.2% of the total vulnerabilities. Other common consequences are client-side XSS and open redirections that affect 10.1% of the pages in total. Finally, 4.2% of the vulnerabilities could lead to cross-site connection hijack of WebSocket and EventSource.

Verification and False Positives. Given the extensive number of reported data flows by Sheriff, we performed a semi-automatic verification as elaborated in §5.4.

In total, the dynamic verification module confirmed about 86% of the data flows (i.e., 202,834 out of 236,427) and eliminated a total of 33,593 FPs across 1,954 webpages and 28 sites. Notably, for the majority of the confirmed flows (i.e., 81%), the verification module successfully validated the vulnerable flow by loading the affected webpage and executing it via Playwright. However, in the remaining 19% of cases (i.e., 38,522 flows), the verifier required executing between one to 41 near-duplicate pages before confirming it. We note that the verifier tests the presence of the data flow also in near-duplicate pages in order to switch the DOM tree with one of the duplicated pages. This is aimed to determine if the same data flow could be observed across various DOM environments during page load, capturing different executions of the program (Cf. §5.4).

We manually confirmed and investigated the reason for false positives by focusing on a random subset. Specifically, we sampled 10 pages per each of the 28 affected sites, which included 5,032 flows in total, and manually inspected them. We observed that a large number of FPs (i.e., 3,951 or 78.5%) occur during the data flow analysis from sources to sinks, and the rest (i.e., 1,081 FPs) occur when checking if a request is triggered at page load or not (i.e., reachability analysis). The former cases happened due to presence of dynamic code evaluation constructs like `eval()` that changed the values of tainted variables, usage of prototype chain with late static binding, generator functions, and inaccurate pointer analysis for property lookups. The latter cases happened due to dynamically called functions, inaccurate pointer analysis, usage of reflection, and dynamic removal of event handlers. Accordingly, verification was critical to eliminate FPs.

Contribution of Dynamic Analysis. We observed that dynamic information plays a crucial role in identifying 67.3% of the discovered request hijacking data flows, as shown in Table 4. First, dynamic taint analysis detected 118.6K vulnerable data flows that were not found by the static

Request Fields								Prevalence		
D	P	B	Q	F	H	S	Total	Flows	Pages	Sites
●	●	○	●	●	○	●	5	2,897	1,103	101
●	●	○	●	○	●	●	5	1,235	235	26
●	●	○	●	○	○	○	3	110	34	11
●	●	○	○	●	○	●	4	88	52	13
●	●	○	○	●	○	○	3	1	1	1
●	●	○	○	○	○	●	3	1,456	391	52
●	●	○	○	○	○	○	2	65	39	5
●	○	●	○	●	○	●	4	1	1	1
●	○	●	○	○	○	○	2	973	159	12
●	○	○	●	○	○	○	2	18	10	2
●	○	○	○	◐	○	●	3	8	6	1
●	○	○	○	○	○	●	2	672	118	10
◐	◐	○	●	○	●	●	5	2	1	1
◐	◐	○	○	○	○	●	3	5	4	1
◐	○	◐	○	○	○	●	3	10	2	2
◐	○	○	○	○	○	○	1	564	95	9
○	●	●	●	○	○	○	3	8	6	1
○	●	○	●	○	○	●	3	1	1	1
○	●	○	○	○	○	●	2	3	3	1
○	●	○	○	○	○	○	1	342	124	13
○	◐	○	●	○	●	○	3	3	1	1
○	◐	○	○	●	○	●	3	1	1	1
○	◐	○	○	○	○	●	2	15	1	1
○	◐	○	○	○	○	○	1	9,640	2,024	219
○	○	●	●	○	○	○	2	92	47	6
○	○	◐	○	○	○	○	1	95,601	12,187	981
○	○	○	◐	●	○	○	2	215	74	5
○	○	○	◐	○	○	○	1	88,009	9,356	747
○	○	○	○	○	●	○	1	799	400	36

Legend: D= Domain; P= Path; B= Body; Q= Query; F= Fragment; H=Headers; S=Scheme;
○ = Not Controllable (00); ◐ = Partial Control (01); ● = Full Control (10);

TABLE 5: Anatomy of client-side forgeable requests. The table shows 29 distinct request patterns ordered by the degree of control (descending).

analyzer, i.e., data flow paths containing only dynamic edges. Second, it aided static analysis in identifying 17.9K additional data flows by patching missing HPG edges necessary for vulnerability detection (i.e., mixed data flow paths). However, Table 3 highlights a key challenge of pure dynamic analysis: the large size of reported taint flows, the majority of which were not under attacker control (Cf. Table 10). Conversely, static analysis was able to detect 66.2K data flows. Therefore, a combination of dynamic and static analysis can be advantageous. Dynamic analysis enhances static analysis by supplementing HPG edges (e.g., call graph), while static analysis helps eliminate spurious taint flows that are not controllable by attackers, e.g., due to input validation.

6.3. Anatomy of Hijacked Requests

In this section, we examine the extent of manipulation an attacker can exert on the hijacked requests of Table 4, as the specific forgeable field(s) and the degree of control an attacker possesses over them may affect the potential risk and severity of vulnerabilities. We used Sheriff to extract the vulnerable lines of code, examined the code stack trace and semantics, and characterized the request anatomy as a binary pattern, encoding information about the type and number of request fields that could be manipulated, as well as the type of control in each field. As a result, we identified 29 distinct forgeable request patterns. Table 5 summarizes our findings.

Type of Control. Our analysis revealed that 80% of the forgeable request fields are fully controllable, allowing the attacker to overwrite their values entirely. In the remaining cases, the attacker has partial control over specific parts of the field, such as one or more parts of query parameters, hash fragment, or body, but not complete control.

Forgeable Request Field. The severity of the vulnerability can be influenced by the type of manipulable field. For example, we found that in 8,105 forgeable requests of 161 sites, the attacker can manipulate the domain, and request hijacking could be used to perform cross-origin attacks (e.g., leakage of CSRF tokens). We grouped requests in seven categories based on the specific field(s) being manipulated, where each request may fall into multiple groups. Our analysis uncovered that the most frequent types of manipulable fields are request body and query parameters, accounting for over 47% and 45% of the forgeable requests respectively. Additionally, the forgeability of domain and path fields in ∼11.8% of the requests is concerning. Finally, we observed that other request fields like headers, hash fragment and scheme are forgeable in about 5.7% of the cases.

Degree of Manipulation. We found that the number of concurrently manipulable request fields varies from one to five out of a total of seven forgeable fields. For example, for 2,897 forgeable requests from 101 sites, the attacker has full control over all URL fields but lacks control over request headers and body. In contrast, in 95K requests on 981 sites and 88K requests on 747 sites, the attacker can manipulate only the request body and query parameters, respectively. We observed that in the majority of the hijacked requests (i.e., 97%), only one or two fields can be manipulated. However, such ad-hoc manipulation capability can still lead to severe consequences (Cf. §6.4).

Request Method. We found that in ∼26% of the cases, the request method is controllable by the attacker. Conversely, for the non-controllable cases, we observed that 51.5% utilized the GET method, while ∼34% opted for POST. The remaining 14.5% employed other state-changing methods such as PUT and DELETE.

6.4. Exploitations

We manually examined the exploitability of the identified vulnerabilities by a web attacker. Due to the large number of confirmed data flows (202K across 961 sites), manual exploit creation for all was impractical. Instead, our goal was to demonstrate exploitability by focusing on a random subset. To ensure comprehensiveness, we aimed to maximize the coverage of our testing across various sites. Therefore, we randomly selected two vulnerable pages from each of the 961 affected sites. We used our analysis of §6.3 to prioritize testing efforts by focusing first on requests with a higher degree of manipulation across various types of client-side requests. Then, we confirm the forgeability of requests and

look for their use in attacks that we presented in §4. For each attack scenario, we conducted specific checks. For example, we looked for server-side endpoints that could lead to security-sensitive state changes (e.g., modifying user settings) for client-side CSRF. For information leakage, we examined the request body for the presence of sensitive data like PII, authorization keys, and CSRF tokens. Furthermore, For WebSocket and EventSource, we check whether we can establish arbitrary connections to attacker-controlled endpoints. Finally, for open redirect and client-side XSS attacks, we assessed the susceptibility of top-level requests to arbitrary redirections and improper validation of `javascript` URIs, respectively. In doing so, we limited our tests exclusively to user accounts that we created on those sites, and excluded testing requests and functionalities where we could not control the impact (e.g., publicly accessible content).

Table 6 summarizes the attacks we uncovered during our investigation. In total, we created 67 proof-of-concept exploits across 49 websites, with far-reaching consequences like CSRF, client-side XSS, open redirections and leakage of sensitive information across various popular platforms and functionalities. Notably, we discovered an account takeover exploit in the Starz movie streaming service, user VM deletion in Microsoft Azure, arbitrary redirection in Google DoubleClick and VK, manipulation of account settings in DW and BBC, tampering of job applications in Indeed, data exfiltration through WebSocket and EventSource hijacks in JustWatch and Forbes, CSRF on `PushManager` subscriptions in Reddit, persistent client-side DoS on push notifications in Yoox shopping website, and finally client-side XSS in TP-Link, to name only a few examples. Among these, a total of 33 exploits across 24 sites belong to new vulnerability types presented in our work. We refer interested readers to §A.2 for case studies of the confirmed attacks.

In the other reviewed cases, we were unable to create exploits or impact was low. However, achieving completeness in the manual search for exploits is a challenging task, requiring extensive knowledge of each specific application for target endpoint identification, request semantics, and the contexts where an attacker could inject payloads as well as strict adherence to ethical standards, e.g., excluding testing of cases not in compliance with vulnerability disclosure programs. For these reasons, automating exploitation is challenging.

	Vulnerability	CSRF	XSS	WS Hijack	SSE Hijack	Inf. Leak	Open Red.	DoS	Total
⊞	Forge. Aysnc Req. URL	7/6	-	-	-	12/7	-	-	19/13
⊞	Forge. Aysnc Req. Body	4/4	-	-	-	-	-	-	4/4
✚	Forge. Aysnc Req. Header	1/1	-	-	-	-	-	-	1/1
✚	Forge. Push Req. URL	-	-	-	-	2/2	-	2/2	4/4
✚	Forge. Push Req. Body	1/1	-	-	-	-	-	-	1/1
✚	Forge. EventSource URL	-	-	-	1/1	1/1	-	-	2/2
✚	Forge. WebSocket URL	-	-	2/2	-	4/2	-	-	6/4
✚	Forge. WebSocket Body	2/1	-	2/2	-	-	-	-	4/3
	Forge. Location URL	1/1	3/3	-	-	-	7/6	-	11/7
✚	Forge. Window Open URL	1/1	6/6	-	-	-	8/8	-	15/10
	Total	17/15	9/9	4/4	1/1	19/12	15/14	2/2	**67/49**

Legend: M/N= M exploits across N sites.

TABLE 6: Summary of exploitations for client-side request hijacking vulnerabilities. Rows marked with ✚ and ⊞ represent new vulnerabilitiy types and variants with a new API or exploitation, respectively.

7. Defenses

This section addresses RQ3, where we review and assess the adoption and efficacy of existing countermeasures against client-side request hijacking vulnerabilities. We systematically surveyed academic literature (i.e., [1–3, 6, 9, 10, 12, 18–20, 23, 30, 44, 56, 75–80]), W3C specifications [81], and OWASP CheatSheet Series [14], looking for classical anti-CSRF countermeasures and those defenses that can mitigate client-side request hijacking. Table 7 summarizes our findings. In total, we identified 10 distinct request forgery defenses, that we grouped into two broad categories based on the party that enforces them (i.e., web application or the browser). For each defense, the table represents whether the defense is effective against client-side request hijacks, there is built-in browser support to enforce it, it is enabled-by-default, whether it requires correct configuration (when offered built-in by the browser), and finally whether it requires custom implementation by web application developers. The rest of this section discusses adoption and efficacy of each defense.

Traditional Mechanisms. CSRF attacks can be mitigated by employing various countermeasures, such as anti-forgery tokens [1, 2, 9, 14, 18, 20], CORS preflight requests [20], `Origin/Referer` [1, 4, 14, 84] header checks, and SameSite cookies [10]. Our measurement in Table 7 shows that these countermeasures are well adopted. For example, we found that 130,359 of the 202K forgeable requests (Cf. Table 4) include a token in the request body or header. Among these, 116,002 cases featured a token name containing 'csrf' or 'xsrf', indicating it was an anti-forgery token. Then, when looking at JavaScript code, we observed that developers explicitly included `Origin/Referer` headers in 42,310 same-site requests. Finally, we observed that 3,751 vulnerable pages (out of 17.8K) use SameSite cookies with `Lax` or `Strict` policies.

While these defenses are necessary to prevent classical request forgery attacks (assuming correct implemention), they are not sufficient to prevent client-side hijack of requests, because JavaScript programs and web browsers include these tokens, headers, and cookies in same-site requests. Also, header-based approaches like CORS are limited to constraining cross-origin requests, but not same-origin requests initiated from the client-side code, thus ineffective against CSRF exploitations. In addition, when the request body contains sensitive information, attackers can hijack the request and reroute it to their own domains, thereby setting the CORS response headers to their advantage.

Input Validation. Robust input validation can ensure data integrity and reliability by requiring untrusted inputs to conform to specific, expected formats [82], preventing malicious

Category	Defense						References	# Pages	# Sites
Custom (Application)	Input Validation	●	◐	○	○	●	[12, 23, 30, 32, 82, 83]	125,738	7,021
	CSRF Tokens*	○	○	○	○	●	[1, 4, 9, 14, 18, 20, 56]	32,925	7,692
	Fetch MetaData*	◐	●	○	●	●	[80]	13,873	910
	Origin/Ref. Headers*	○	●	○	○	●	[1, 10, 14, 20]	9,922	1,745
Policy-based (Browser)	Cross-Origin Resource Sharing	○	●	●	●	○	[35]	284,984	8,741
	SameSite Cookies	○	●	●	●	○	[10, 19, 34, 76]	69,865	5,621
	Content Security Policy	◐	●	○	●	○	[25, 26, 31, 44]	25,799	4,616
	Cross-Origin Opener Policy	◐	●	○	●	○	[27, 80]	6,581	231
	Cross-Origin Embedder Policy	◐	●	○	●	○	[37]	3,314	96

Legend: = Effective; = Built-in Browser Support; = Enabled-by-Default; = Require Configuration; = Require Implementation; ○ = Not Applicable; ◐ = Partially Applicable; ● = Fully Applicable; *= Server-side enforced.

TABLE 7: Summary of existing defenses and their protective coverage against client-side hijacks. The table shows the adoption rate of the various defense mechanisms in the wild. For rows marked with *, the adoption rate only reflects the explicit inclusion of headers/tokens in the client-side code.

inputs reaching request-sending instructions. Accordingly, we identified and analyzed secure and insecure input validation patterns and practices that is employed by websites in the wild against request hijacking attacks as described below.

First, to identify insecure input validation code patterns, we analyzed vulnerable data flows discovered in §6.2, and extracted the underlying reason why the flow was marked as vulnerable, focusing on the presence of insufficient, missing or logically flawed input validation checks. Table 8 summarizes our findings, where we grouped the checks into eight different categories. Our analysis uncovers that ∼47% of vulnerable data flows do not have any input validation checks, suggesting that developers are largely unaware of risks associated with controlling client-side requests. Furthermore, over 13.8% of the cases rely solely on a variety of trivial checks, such as length and type checks, and 24.9% use string operations to search for existence of trusted domains in URLs or check different URL fields, which is insufficient, e.g., the check for presence of `benign.com` can be trivially bypassed if the attacker uses the payload `benign.com.evil.com`. Similarly, checks that only test partial URL fields, such as path or query parameters are insufficient, because attackers may be able to forge the request domains, or overwrite query parameters with parameter pollution [85]. We observed that a different group of data flows (11.1%) apply a combination of these checks simultaneously.

Then, about 2.7% of the data flows contain validation checks that compare two different attacker-controlled values with one another, e.g., a query parameter value, used to generate an asynchronous request, is compared with `window.name`, suggesting that developers treat `window` properties as trusted values that can be safely used in sensitive operations. Finally, less than 0.4% of the input validation checks exhibit logical flaws, where the tested condition always evaluates to true, which could indicate a potential gap between the semantics of the JavaScript language and the developers' comprehension. Also, we examined the input validation implemented on various data flows within the same webpage and across different pages of a site, and we observed at least two distinct types of input validation in 699 pages and 412 sites, respectively, which may suggest the presence of multiple developers' implementations and differences in their approaches to input validation.

Then, we also examine secure patterns that can hinder request hijacking, both intentionally and unintentionally. Specifically, out of the ∼3.3M taint flows that Foxhound$^+$ discovered (Cf. Table 3), Sheriff marks only ∼118K of them as vulnerable, and discards the rest (∼3.2M) due to a variety of (preventive) program behaviours, e.g., validity checks, duplicate executions of the same data flow, and re-assignment of constant values to sources like URL fragment. We used static analysis to examine more closely the reasons why these 3.2M requests were not vulnerable, and we grouped them into seven categories (Cf. Table 10 of §A.3).

In total, Sheriff identified 1,104,104 data flows across 125,738 webpages and 7,021 sites that implement robust input validation against request hijacking. We found that overwriting attacker-controlled sources with variable assignments and strict equality / whitelist comparisons are the most common type of input validation, which prevents request hijacking, with a total of 653K and 432K instances across 3,935 and 2,824 sites, respectively. Then, contrary to these intentional checks, we found that other program behaviours may prevent request hijacking too. For example, the most frequent reason for the non-vulnerability of a taint flow was its sole reliance on the domain or path of the webpage to generate outgoing requests, because modifying the domain or path of the top-level URL by the attacker would result in the victim accessing a different webpage altogether.

Content Security Policy (CSP). CSP [25, 26] can limit the impact of request hijacking when attackers can forge the URL of requests. In these cases, CSP `connect-src` directive [36] can be used to constrain endpoints for asynchronous requests, EventSource and WebSockets to trusted domains, preventing sensitive data exfiltration to other domains. We found that a correct configuration of CSP could mitigate information leakage and XSS exploitations in 58.7% of the request hijacking data flows. However, we observed that only 7.6% of the webpages in our dataset deploy a CSP using this directive, including 1,265 pages with vulnerable data flows. While CSP can mitigate information leakage, it does not prevent hijacking requests for CSRF attacks (i.e., same-site request endpoints, or forging request body).

Cross-Origin Opener Policy (COOP). When attackers use

Check	Instances	Flows	Pages	Sites
No Check	S	95,321	8,876	709
Substring Search	S.indexOf('benign.com') > 0	62,495	3,950	285
	S.startswith('benign.com')	11,448	821	83
	S.includes('benign.com')	2,024	145	32
Not Null	typeof S !== 'undefined' && S!== null	32,002	2,616	194
Length	S && S.length > 0	13,995	1,023	83
Empty String	S !== ''	6,179	638	156
Comparison of Forgeable Params	QUERY(Q, S)=== window.name	4,776	445	65
	QUERY(Q, S)=== loc.hash.substr(i, j)	556	39	3
	postMessage(S) === loc.hash.substr(i, j)	102	10	1
URL Fields Check	PATH(S) == 'index.php'	1,199	92	10
	QUERY(Q, S) === 'benign'	402	33	5
Faulty Conditionals (Always True)	if (S === 'b1.com' || 'b2.com')	130	11	3
	!! 'benign.com' == !! S	629	40	3
	S !== undefined + (S === 'benign.com')	14	6	1
	intersection(['b1.com', 'b2.com'], [S]) !== []	21	5	2
	S.length $\leq$ Math.min() + CONST	5	2	1

Legend: S= Source; QUERY(Q, URL)= query parameter Q in URL
PATH(URL) = URL path.

TABLE 8: Types of input validation checks in vulnerable data flows.

`window.open()` to open vulnerable target pages in a new window, COOP [27] can be used to isolate the browsing context to same-origin documents. For example, if an honest, cross-origin page with COOP is opened in a new window, the malicious opening page will not have a reference to it, preventing attackers to set the window name, or send postMessages to the new window, which in turn prevents the forgery of requests generated by these inputs. We found that about 7% of the request hijacking data flows could be mitigated by COOP, as they rely on window name, document referrer and postMessages to provide program inputs. However, we observed that only ∼1.9% of webpages in the wild implement COOP, and, strikingly, none of the webpages exhibiting request hijacking data flows had adopted this policy, calling for increased awareness about COOP.

Cross-Origin Embedder Policy (COEP). COEP [37] controls embedded cross-origin resources in a webpage. Developers can use the COEP `require-corp` policy to restrict fetched resources to either the same origin or a set of explicitly marked cross-origin resources. As such, COEP can constrain the `fetch()` API to trusted domains, mitigating the impact of 5.3% of the total request hijacks. We observed that only ∼1% of the webpages in our dataset use the `require-corp` policy, including 141 pages with vulnerable data flows across 32 sites.

Fetch MetaData. These are a series of HTTP request headers [80, 86] that send additional provenance meta data about the request, such as the context it originated from. Websites can use this information to implement policies that block potentially malicious requests. While Fetch MetaData headers are automatically included in client-side requests by JavaScript programs, they can still restrict exploitations. For example, websites can use the `Sec-Fetch-Mode` header with the `navigate` option to restrict top-level requests exclusively for page navigation, and block request hijacking attacks that trigger state changes [10]. We observed that these headers are present in 67,221 requests across ∼9% of websites including 85 vulnerable sites.

8. Related Work

Request forgery vulnerabilities have a long history and have been the the subject of numerous research endeavors in the past, e.g., SSRF [79, 87], CSWSH [49–51], CSRF [2, 4, 56, 78], and client-side CSRF [12, 13]. Due to their nefarious consequences, there is a large body of research dedicated to developing defense mechanisms against them (e.g., [1, 3, 9, 10, 18–20, 75, 76]). Closely related to our work, multiple studies considered the hijack of HTML tags such as scripts [30, 54] and iframes [88]. In contrast to HTML tags, our study focuses on JavaScript APIs that allow creating requests. Instead of hijacking requests, previous research also explored injecting new requests thorough DOM manipulations and dangling markup injections [89–91]. Other works studied request forgery vulnerability detection techniques leveraging both manual and (semi-)automated approaches, e.g., static and dynamic analyses [2, 12], ML-based solutions [56], and systematic manual inspection [4, 77]. Our study complements the missing pieces of these works by extending client-side CSRF (i.e., [12]) and proposing client-side variants of classical request forgery attacks, quantifying their prevalence and impact in the wild.

An orthogonal line of related work explored various program analysis techniques for vulnerability discovery, such as static analysis [63, 64, 66, 92], dynamic analysis [2, 23, 30–32], and hybrid approaches [12, 44, 53, 93, 94]. Pertaining closely to our work, several research efforts studied client-side input validation flaws [12, 23, 30, 32, 54, 83, 94]. For example, Klein et. al. [23] used a combination of dynamic taint tracking and symbolic string analysis to study the robustness of custom sanitization functions in the wild. Steffens et. al. assessed the prevalence of persistent [54] and postMessage-based [32] XSS. Other works studied the presence of script-less attacks in JavaScript programs using both static and dynamic approaches, e.g., DOM Clobbering [44], mutation-based XSS [95] and script gadgets [31]. Our work aligns with these efforts by leveraging and combining common program analysis techniques. However, in contrast to these works, we focus on developing a custom-tailored detection technique and tool for client-side request hijacking to study such vulnerable program behaviors on the Web platform.

9. Discussion and Conclusion

In this section we summarize the main findings of our study and discuss their wider implications.

Client-side CSRF Only Tip of the Iceberg. In this paper, we have shown that client-side CSRF is only one facet of the larger problem of request hijacking in web applications. In fact, a considerable fraction of request hijacking data flows that we discovered (36.1%, i.e., 73.3K out of 202K) as well as more than half of the exploits that we created (Cf. §6.4) leverage the new vulnerability types and variants which

have not been considered by previous works on client-side CSRF [12–14]. For example, we observed that over 21% of forgeable data flows affect the `sendBeacon` API [17].

Request Hijacking Data Flows are Ubiquitous. Request hijacking data flows are pervasive in today's web, affecting over 9.6% of the websites in the wild and about 5.2% of the tested webpages in our dataset. Our measurement provides only a lower-bound estimate of the vulnerable data flows as we limited our tests to 50 unique pages of each website.

Request Hijacking has Diverse Consequences. Our work uncovers the diverse range of security implications resulting from client-side request hijacking, where each vulnerability could be exploited in multiple ways depending on the affected request type and API, amplifying the associated risks. For example, we show that the hijack of asynchronous requests not only results in client-side CSRF, as outlined in previous research [12], but also exposes the risk of information leakage, e.g., when attackers gain control over the endpoint of a request that contains sensitive information in its body. We observed that over 41% of the exploitable data flows (Cf. §6.4) could lead to client-side XSS and information leakage, and 25.3% and 22.4% lead to client-side CSRF and open redirections, respectively.

Existing Defenses Necessary but Insufficient. The analysis of existing countermeasures (§7) suggests that they are a necessary protection mechanism to prevent classical attacks (e.g., CSRF), but do not provide a complete protective coverage as each can only mitigate a fraction of the resulting attacks. For example, CSP does not mitigate over 41% of the XSS and information leakage exploitations of request hijacking, and is ineffective against CSRF exploitations, and COOP cannot prevent ∼93% of the discovered request hijacking vulnerabilities. In the absence of a full-fledged browser-level defense, developers have to be particularly careful when choosing or implementing a countermeasure, in order to balance security with usability. For example, over 9.6% of the applications have insufficient, missing or logically flawed input validation checks when offering their functionality. Therefore, we believe proper input validation should be the primary means to defend against these attacks, e.g., by limiting request API parameters to a predefined allow-list (i.e., the input is only used to pick one from the allow-list). One interesting future direction is exploring input validation mechanisms built into browsers such as the new Sanitizer API [96].

Open Science. To support the future research effort, we publicly release Sheriff[2] and Foxhound+[3], that can be used to detect and study request hijacking vulnerabilities at scale.

Ethical Considerations. Our experiments on live websites do not target any real user. Tests requiring state-changing operations (e.g., changing profile settings) are restricted to accounts that we created on those sites exclusively, and we also excluded testing functionalities where we could not control the request impact (e.g., public comments and posts). Throughout the testing process, we strictly adhered to best practices and guidelines outlined by websites' vulnerability disclosure programs on platforms like HackerOne [97] and BugCrowd [98], whenever such programs were available.

The vulnerabilities identified in this paper affect 961 websites including 49 sites for which we created exploits. In June 2023, we initiated the process of notifying the affected parties, adhering to the best vulnerability disclosure practices [99, 100]. We prioritized our reports based on their severity, where we first focused on sites with a known exploit. We sent an initial notification including a detailed description of the vulnerability or a proof-of-concept exploit, and follow up with additional reminders every month to increase the remediation rate. At the time of writing the paper, we have notified all of the 49 sites for which we created an exploit at least once, out of which 37 sites have already confirmed the issues and 23 sites patched them, such as Microsoft Azure, Reddit, Forbes, Google DoubleClick, Starz and Indeed. For the remaining vulnerable data flows, we need to contact 912 sites. In July 2023, we sent the first reports to 62 of them (11 confirmed so far). To contact the remaining sites, we seek the assistance of our national CSIRT.

2. https://github.com/SoheilKhodayari/JAW
3. https://github.com/SAP/project-foxhound

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] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *ACM CCS*, 2008.
[2] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow, "Deemon: Detecting CSRF with dynamic analysis and property graphs," in *ACM CCS*, 2017.
[3] M. Johns and J. Winter, "RequestRodeo: Client side protection against session riding," 2006, https://www.owasp.org/images/4/42/RequestRodeo-MartinJohns.pdf.
[4] A. Sudhodanan, R. Carbone, L. Compagna, and N. Dolgin, "Large-scale analysis & detection of authentication cross-site request forgeries," in *IEEE EuroS&P Symposium*, 2017.
[5] J. Burns, "Cross site reference forgery: An introduction to a common web application weakness," in *Information Security Partners, LLC*, 2005.
[6] K. Käfer, "Cross site request forgery," in *Hasso-Plattner-Institut, Technical report*, 2008.
[7] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," in *ACM SIGOPS Operating Systems Review*, 1988.
[8] D. Ferguson, "Netflix Cross Site Request Forgery Vulnerability," *SecList Full Disclosure Mailing List*, 2006, https://seclists.org/fulldisclosure/2006/Oct/316.
[9] W. Zeller and E. W. Felten, "Cross-site request forgeries: Exploitation and prevention," in *Princeton University*, 2008.
[10] S. Khodayari and G. Pellegrino, "The state of the samesite: Studying the usage, effectiveness, and adequacy of samesite cookies," in *IEEE S&P Symposium*, 2022.
[11] (2019) Critical CSRF Vulnerability on Facebook. https://www.acunetix.com/blog/web-security-zone/critical-csrf-vulnerability-facebook/.
[12] S. Khodayari and G. Pellegrino, "JAW: studying Client-side CSRF with hybrid property graphs and declarative traversals," in *USENIX Security Symposium*, 2021.
[13] (2018) Client-side CSRF. https://www.facebook.com/notes/facebook-bug-bounty/client-side-csrf/2056804174333798/.

[14] OWASP cross-site request forgery prevention cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html.
[15] XMLHttpRequest API. https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest.
[16] "Fetch Living Standard," https://fetch.spec.whatwg.org.
[17] "Beacon, W3C Working Draft," 2023. [Online]. Available: https://www.w3.org/TR/beacon/
[18] J. Wilander, "Advanced csrf and stateless anti-csrf," 2012.
[19] P. D. Ryck, L. Desmet, W. Joosen, and F. Piessens, "Automatic and precise client-side protection against CSRF attacks," in *ESORICS*, 2011.
[20] X. Likaj, S. Khodayari, and G. Pellegrino, "Where we stand (or fall): An analysis of csrf defenses in web frameworks," in *RAID Symposium*, 2021, pp. 370–385.
[21] Bitnami Application Catalog. https://bitnami.com/stacks.
[22] Project Foxhound. https://github.com/SAP/project-foxhound.
[23] D. Klein, T. Barber, S. Bensalim, B. Stock, and M. Johns, "Hand Sanitizers in the Wild: A Large-scale Study of Custom JavaScript Sanitizer Functions," in *IEEE EuroS&P*, 2022.
[24] Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/.
[25] M. West, "Content Security Policy Level 3," *W3C Working Draft*, 2022, https://w3c.github.io/webappsec-csp/.
[26] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy," in *ACM CCS*, 2016, pp. 1376–1387.
[27] Cross-Origin Opener Policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy.
[28] J. Schwenk, M. Niemietz, and C. Mainka, "Same-Origin Policy: Evaluation in Modern Browsers," in *USENIX Security Symposium*, 2017.
[29] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song, "Towards a formal foundation of web security," in *IEEE CSF*, 2010.
[30] S. Lekies, B. Stock, and M. Johns, "25 million flows later: large-scale detection of DOM-based XSS," in *ACM CCS*, 2013.
[31] 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.
[32] M. Steffens and B. Stock, "PMForce: Systematically Analyzing postMessage Handlers at Scale," in *CCS*, 2020.
[33] window.open() API. https://developer.mozilla.org/en-US/docs/Web/API/Window/open.
[34] "Cookies: HTTP State Management Mechanism," 2020. [Online]. Available: https://tools.ietf.org/html/draft-ietf-httpbis-rfc6265bis-05
[35] Cross-Origin Resource Sharing. https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.
[36] CSP connect-src Directive. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/connect-src.
[37] Cross-Origin Embedder Policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Embedder-Policy.
[38] "HTML Living Standard," 2023. [Online]. Available: https://html.spec.whatwg.org/
[39] "XMLHttpRequest Living Standard," https://xhr.spec.whatwg.org/.
[40] "Push API Specification, W3C Working Draft," 2023. [Online]. Available: https://www.w3.org/TR/push-api/
[41] "WebSockets Living Standard," 2023. [Online]. Available: https://websockets.spec.whatwg.org/
[42] "WHATWG Specifications," https://spec.whatwg.org/.
[43] "W3C Standards and Drafts," https://www.w3.org/TR/.
[44] S. Khodayari and G. Pellegrino, "It's (dom) clobbering time: Attack techniques, prevalence, and defenses," in *IEEE S&P Symposium*, 2023.
[45] "Push API: CSRF on PushManager Subscriptions." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Push_API
[46] K. Subramani, J. Jueckstock, A. Kapravelos, and R. Perdisci, "Sok: Workerounds-categorizing service worker attacks and mitigations," in *IEEE EuroS&P Symposium*, 2022.
[47] T. Watanabe, E. Shioji, M. Akiyama, and T. Mori, "Melting pot of origins: Compromising the intermediary web services that rehost websites." in *NDSS Symposium*, 2020.
[48] I. Hickson, "Server-sent Events," in *W3C Working Draft*, 2012. [Online]. Available: https://www.w3.org/TR/2012/WD-eventsource-20120426/
[49] C. Schneider, "Cross-Site WebSocket Hijacking (CSWSH)," 2019. [Online]. Available: https://christian-schneider.net/CrossSiteWebSocketHijacking.html
[50] "Cross-Site WebSocket Hijacking." [Online]. Available: https://portswigger.net/web-security/websockets/cross-site-websocket-hijacking
[51] W. Mei and Z. Long, "Research and Defense of Cross-Site WebSocket Hijacking Vulnerability," in *IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*, 2020.
[52] P. Murley, Z. Ma, J. Mason, M. Bailey, and A. Kharraz, "WebSocket Adoption and the Landscape of the Real-Time Web," in *WWW Web Conference*, 2021. [Online]. Available: https://doi.org/10.1145/3442381.3450063
[53] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis," in *NDSS Symposium*, 2007.
[54] 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.
[55] "Exposure of Sensitive Information to Unauthorized Actors in EventSource," 2022. [Online]. Available: https://huntr.dev/bounties/dc9e467f-be5d-4945-867d-1044d27e9b8e/
[56] S. Calzavara, M. Conti, R. Focardi, A. Rabitti, and G. Tolomei, "Mitch: A machine learning approach to the black-box detection of csrf vulnerabilities," in *IEEE EuroS&P Symposium*, 2019.
[57] The WebSocket API. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
[58] 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 Symposium*, 2019.
[59] Playwright browser automation framework. https://playwright.dev/.
[60] Firefox developer tools. https://firefox-dev.tools/.
[61] S. Pletinckx, K. Borgolte, and T. Fiebig, "Out of Sight, Out of Mind: Detecting Orphaned Web Pages at Internet-Scale," in *ACM CCS*, 2021.
[62] M. Henzinger, "Finding near-duplicate web pages: a large-scale evaluation of algorithms," in *ACM SIGIR conference on Research and development in information retrieval*, 2006.
[63] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in *IEEE S&P Symposium*, 2014.
[64] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and Flexible Discovery of PHP Application Vulnerabilities," in *IEEE EuroS&P Symposium*, 2017.
[65] Neo4j. https://neo4j.com/.
[66] T. Brito, P. Lopes, N. Santos, and J. F. Santos, "Wasmati: An efficient static vulnerability scanner for WebAssembly," *Computers & Security*, 2022.
[67] S. Guarnieri and B. Livshits, "GULFSTREAM: Staged Static Analysis For Streaming JavaScript Applications," in *Proceedings of the USENIX conference on Web application development*, 2010.
[68] S. H. Jensen, P. A. Jonsson, and A. Møller, "Remedying the Eval that Men Do," in *ACM ISSTA*, 2012.
[69] K. Gallaba, A. Mesbah, and I. Beschastnikh, "Dońt Call Us, Wéll Call You: Characterizing Callbacks in Javascript," in *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2015.
[70] M. Madsen, B. Livshits, and M. Fanning, "Practical Static Analysis of Javascript Applications in the Presence of Frameworks and Libraries," in *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2013.
[71] S. H. Jensen, M. Madsen, and A. Møller, "Modeling the HTML DOM and Browser API in Static Analysis of Javascript Web Applications," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)*, 2011.
[72] setTimeout global function. https://developer.mozilla.org/en-US/doc

s/Web/API/setTimeout.
[73] Ineo: neo4j instance and version manager. https://github.com/cohesivestack/ineo.
[74] window.name API. https://developer.mozilla.org/en-US/docs/Web/API/Window/name.
[75] S. Lekies, W. Tighzert, and M. Johns, "Towards stateless, client-side driven cross-site request forgery protection for web applications," *SAP Research*, 2012.
[76] A. Czeskis, A. Moshchuk, T. Kohno, and H. J. Wang, "Lightweight server support for browser-based CSRF protection," in *WWW Web Conference*, 2013.
[77] H. Shahriar and M. Zulkernine, "Client-side detection of cross-site request forgery attacks," in *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering*, 2010.
[78] E. Shernan, H. Carter, D. Tian, P. Traynor, and K. Butler, "More guidelines than rules: Csrf vulnerabilities from noncompliant oauth 2.0 implementations," in *DIMVA*, 2015.
[79] B. Jabiyev, O. Mirzaei, A. Kharraz, and E. Kirda, "Preventing server-side request forgery attacks," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, 2021, pp. 1626–1635.
[80] A. Sudhodanan, S. Khodayari, and J. Caballero, "Cross-origin state inference (COSI) attacks: Leaking web site states through xs-leaks," in *NDSS Symposium*, 2020.
[81] W3C Standards and Drafts. https://www.w3.org/TR/.
[82] M. Alkhalaf, T. Bultan, and J. L. Gallegos, "Verifying client-side input validation functions using string analysis," in *ICSE*, 2012.
[83] M. Weissbacher, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "ZigZag: Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities," in *USENIX Security Symposium*, 2015.
[84] F. Kerschbaum, "Simple cross-site attack prevention," in *IEEE SecureComm*, 2007.
[85] M. Balduzzi, C. T. Gimenez, D. Balzarotti, and E. Kirda, "Automated discovery of parameter pollution vulnerabilities in web applications." in *NDSS Symposium*, 2011.
[86] Fetch MetaData Sec-Fetch-Dest Header. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Sec-Fetch-Dest.
[87] G. Pellegrino, O. Catakoglu, D. Balzarotti, and C. Rossow, "Uses and abuses of server-side requests," in *RAID Symposium*, 2016.
[88] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson, "Busting frame busting: a study of clickjacking vulnerabilities at popular sites," *IEEE S&P Symposium*, 2010.
[89] M. Heiderich, C. Späth, and J. Schwenk, "DOMPurify: Client-side protection against xss and markup injection," in *ESORICS*, 2017.
[90] F. Hantke and B. Stock, "HTML violations and where to find them: a longitudinal analysis of specification violations in HTML," in *ACM Internet Measurement Conference*, 2022.
[91] Dangling markup injection. https://portswigger.net/web-security/cross-site-scripting/dangling-markup.
[92] F. Al Kassar, G. Clerici, L. Compagna, F. Yamaguchi, and D. Balzarotti, "Testability tarpits: the impact of code patterns on the security testing of web applications," 2022.
[93] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan, "NAVEX: Precise and scalable exploit generation for dynamic web applications," in *USENIX Security Symposium*, 2018.
[94] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for JavaScript," in *IEEE S&P Symposium*, 2010, pp. 513–528.
[95] 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.
[96] F. Braun, M. Heiderich, and D. Vogelheim, "HTML Sanitizer API," *W3C Draft Community Group Report*, 2023, https://wicg.github.io/sanitizer-api/.
[97] Hackerone. https://hackerone.com.
[98] Bugcrowd. https://www.bugcrowd.com.
[99] 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 Symposium*, 2016.
[100] 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 Symposium*, 2016.
[101] JavaScript Function() constructor. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/Function.

Listing 2: Example client-side request hijacking vulnerability derived from `bbc.com`, which is not captured by JAW's static analysis engine [12].

```
1  var c = {}, i = 0;
2  // handle incoming postMessages
3  window.addEventListener("message", h);
4  function h(e){
5    if(e.origin.indexOf("bbc.com") > -1){
6      i = i + 1;
7      // [...]
8      var d = JSON.stringify({
9        "csrf_token": "xyz-token",
10       "state": {...},
11     });
12     var u = e.data + '/userinfo';
13     c["r" + i] = new
14         Function("httpPostRequest("+ u + "," + d + ")");
15   }
16 }
17 function httpPostRequest(url, body){
18     // [...]
19     navigator.sendBeacon(url, body)
20 }
21 // remember state upon closing the session
22 window.addEventListener("visibilitychange", saveState);
23 function saveState(e) {
24   if (document.visibilityState === "hidden") {
25     for(let j=1; j<= i; j++){
26       c["r" + j]();
27     }
28 }}
```

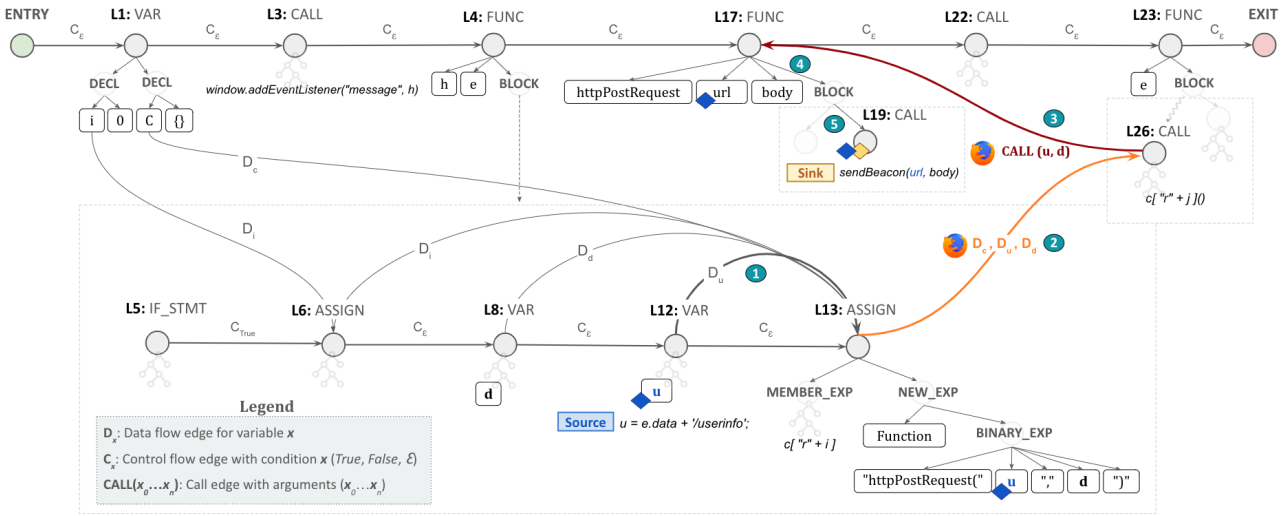
Appendix A.

A.1. TA-HPG Construction and Analysis

In this section, we exemplify the TA-HPG construction and analysis approach presented in Sections 5.2.2 and 5.3 through an example of a real vulnerability derived from `bbc.com` (Cf. §A.2), highlighting the need for dynamic information provided by Foxhound$^+$.

Motivating Example. Listing 2 shows a real example of client-side request hijacking vulnerability derived from `bbc.com`, where the program uses attacker-controlled inputs to specify the endpoint to which an asynchronous HTTP POST request is sent to. In more detail, the code first listens for incoming postMessages (line 3), and then uses the message data to construct a URL (lines 4-12). Afterwards, it creates a closure function using the `new Function()` API [101] by generating a string of the target function call dynamically (lines 13-14). The string contains an invocation of the `httpPostRequest` function in line 17, with parameters being the constructed URL of line 12 (attacker-controlled), and sensitive data of line 8 (i.e., CSRF tokens). Subsequently, it stores the closure function as a property of the global object `c` (line 13). Finally, upon closing the session (line 22), the program uses dynamic property lookups to retreive the closure function stored in object `c` and invokes it (line 26), which in turn sends an HTTP POST request (line 19) to the attacker-controlled endpoint.

Figure 3: Excerpt of the TA-HPG for the example in Listing 2. Connections highlighted in orange and red represent missing PDG and call graph edges that are reconstructed using dynamic taint flows of Foxhound+, which are necessary for vulnerability discovery (steps 1-5). Blue and yellow diamonds attached to nodes represent source and sink semantic types propagated through the TA-HPG. For brevity, not all nodes and edges are shown.



TA-HPG Construction and Traversals. The dynamic JavaScript language features used in Listing 2 present significant challenges for static analysis-based approaches like JAW [12] to capture the aforementioned request hijacking vulnerability. For example, JAW cannot identify the invoked function and its corresponding arguments in line 26. This is due to dynamic property reads/writes on lines 13 and 26, as well as dynamic code generation using `new Function()` on line 14, which makes it difficult to create a comprehensive representation of the program.

Figure 3 presents the TA-HPG that Sheriff generates for the code in Listing 2, which alleviates the missing HPG edges due to the dynamic function calls. In particular, Sheriff uses dynamic taintflows provided by Foxhound+ to add (i) a call edge between the call expression node in line 26 and the function declaration node in line 17, and (ii) PDG data dependency edges from assignment expression in line 13 to call expression node in line 26 for call arguments `u` and `d`. Accordingly, a TA-HPG traversal can now start from the source node in L12, pass through L13, L26, and L17 nodes, and finally reach the sink instruction, who picks up the attacker-controlled values.

A.2. Case Studies

In this section, we present a few manually vetted case studies of the confirmed attacks, which were patched following our vulnerability disclosure. In §2, we presented the request hijacking vulnerability we discovered in Microsoft Azure. In this section, we present additional case studies based on coverage of various attack types and popularity of the affected parties.

Indeed. We found that the JavaScript code reads the value of a query parameter `r` through `location.search` API, and use it as the path of an asynchronous POST request endpoint without proper validation. Attackers could leverage this behaviour to forge arbitrary requests toward state-changing server-side endpoints. For example, we created a client-side CSRF exploit that enables attackers to modify the details of user job applications or withdraw it without their knowledge.

Reddit. We found that Reddit relies on URL parameters to construct the endpoint of a push subscription request. The request body contained information that the application needs to send a push message such as a push endpoint and the encryption key, which is security-sensitive. Attackers can steal this information by hijacking the push subscription request (through manipulation of URL parameters) and diverging it to the domain they control. Consequently, the leaked endpoint and encryption key can be abused to send malicious messages to the victim's browser.

Starz. We discovered that the endpoint of a client-side redirect request originates from `window.name`, which is attacker-controlled. The JavaScript program generated this request via `window.open()` API and employed the current window as the browsing target. This allows attackers to hijack and exploit the request with `javascript` URIs and transform the redirection to client-side XSS, which in turn enables the hijack of session id and compromise of user accounts.

Forbes. This is a news websites that uses server-sent events (SSE) for updating the user's news feed. Unfortunately, it was possible for attackers to manipulate the EventSource URL and forward it to a malicious domain they control, because the JavaScript program used a query parameter value as the EventSource URL with little-to-no input validation checks. This enabled attackers to obtain SSE hijacking and send malicious events to user's browsers.

BBC. We found that the endpoint of an asynchronous POST request originates from a broadcasted postMessage, which

attackers can control. As such, attackers can manipulate the request endpoint and set it to a domain they control, enabling them to steal the CSRF token that is included in the request body by the JavaScript prorgram. Subsequently, the leaked token can be used to perform CSRF attacks. For example, we created an exploit to modify user notification settings.

JustWatch. We discovered that the JavaScript program retrives the value of the URL hash fragment and use it as the endpoint of a WebSocket, which is used to perform the initial handshake for socket connection. Attackers can redirect this request to a malicious domain they control, resulting in leakage of messages communicated by the user.

Yoox. This is a shopping cart application that employs push notifications as a means to promote clothing products. However, we identified a vulnerability affecting this functionality wherein the endpoint of push subscription requests can be manipulated through URL parameters. This manipulation grants attackers the ability to launch DoS attacks on push notifications by changing the subscription endpoint to an invalid value. Fortunately, affected users can mitigate the DoS and regain access to the intended functionality by resetting the browser notification permissions, which restores the proper operation of the push notification system.

TP-Link. We found a client-side request whose endpoint originated from a query parameter named `url`, which affects the TP-link page preview functionality. The program retrieves the parameter value and then redirects the webpage URL to the read value. As this request is top-level, the hijacked request could be exploited to acheive client-side XSS by abusing `javascript` URIs, as the JavaScript program does not perform proper input validation.

A.3. Additional Evaluation Details

	Vulnerability	CSP	COOP	COEP	CORS	SameSite Cookies	Input Validation	CSRF Tokens	Origin Header	Fetch MetaData
⊞	Forge. Async Req. URL	◐	◐	◐	○	○	●	○	○	◐
⊞	Forge. Async Req. Body	○	◐	○	○	○	●	○	○	◐
✚	Forge. Async Req. Header	○	◐	○	○	○	●	○	○	○
✚	Forge. Push Req. URL	◐	◐	○	○	○	●	○	○	◐
✚	Forge. EventSource URL	◐	◐	○	○	○	●	○	○	◐
✚	Forge. WebSocket URL	◐	◐	○	○	○	●	○	○	◐
✚	Forge. WebSocket Body	○	◐	○	○	○	●	○	○	◐
	Forge. Location URL	○	◐	○	○	○	●	○	○	◐
✚	Forge. Window Open URL	○	◐	○	○	○	●	○	○	◐

Legend: ● = Effective; ◐ = Partially Effective; ○ = Not Effective.

TABLE 9: Protective coverage of existing defenses over different client-side request hijacking variants. Defenses with partial effectiveness mitigate only manipulation of specific APIs, or certain exploitations of the vulnerability.

Property	Instances	Flows	Pages	Sites
Infeasible Source Manipu.	SP is URL domain	1,152,266	116,441	3,249
	SP is URL path	911,897	95,269	2,657
Reassignment to Source	SP = constant	627,460	68,251	3,358
	SP: fragment string replace	21,709	3,067	1,502
	SP: fragment object assign	4,092	666	434
Whitelist / Equality Check	SP === constant	367,024	49,089	2,294
	SP: postMessage origin check	40,185	7,634	965
	SP includes a set of constants	15,032	2,541	367
	arrayConstants.includes(SP)	10,022	899	610
Duplicate Function Calls	SP flow executed $\geq$1 times	8,743	1,228	202
Length Check	Length(SP) === 1	3,560	1,155	501
Type Check	typeof SP === "number"	12,121	2,001	1,328
	typeof SP === "boolean"	1,667	1,001	542
	SP instanceof Date	789	300	91
	SP is JSON && valid(SP)	443	235	55
Benign Control / Taint	SP taints request fragment only	97,528	17,802	1,029
	SP taints request scheme only	44,209	10,512	836

Legend: SP= Source Parameter.

TABLE 10: Summary of program behaviours that can eliminate unique client-side request hijacking vulnerabilities.

Figure 4: Example of request fields as in Table 5.

Scheme	*Domain*	*Path*	*Query*	*Fragment*

```
https://example.com/pref/change.php?r=/profile#!/user/log
```

Headers

```
content-type: text/html; charset: utf-8
x-api-key: "wxyz12"
…
```

Body

```
{ … }
```

Category	JavaScript Sink
Request Hijacking [12, 17, 33, 40, 44, 48, 51]	✪ `navigator.sendBeacon(T1, T2)`
	`fetch(T1, T2)`
	`XMLHttpRequest.open(T)`
	`xhr.send(T)`
	`xhr.setRequestHeader(T1, T2)`
	✪ `new WebSocket(T)`
	✪ `socket.send(T)`
	✪ `new EventSource(T)`
	✪ `PushManager.subscribe(T)`
	`window.open(T)`
	`location.href = T`
	`location.replace(T)`
	`location.assign(T)`
Code Execution [23, 30, 54, 94]	`eval(T)`
	`new Function(T)`
	`setInterval(T)`
	`setTimeout(T)`
	`script.text = T`
	`script.src = T`
	`script.innerHTML = T`
Markup Injection [31, 44, 95]	`document.write(T)`
	`document.writeln(T)`
	`elm.innerHTML = T`
	`elm.outerHTML = T`
	`elm.insertAdjacentHTML(T)`
	`elm.insertAdjacentText(T)`
State Manipulation [44, 54]	`document.cookie = T`
	`localStorage.setItem(T)`
	`sessionStorage.setItem(T)`
PostMessage Spoofing [32]	`postMessage(T)`

Legend: T_i= Tainted Variable.

TABLE 11: Summary of primitive JavaScript sinks supported by Sheriff. Rows marked with ✪ show APIs for which we implemented extra instrumentation in Foxhound+.

Appendix B.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

Starting from a review of browser APIs and Web specifications, the paper presents a systematization of client-side request hijacking vulnerabilities. It identifies 10 distinct vulnerability variants, including seven new ones, and discusses their security implications. The paper introduces Sheriff, a tool that combines static analysis with dynamic taint tracking to detect vulnerable data flows from attacker-controllable inputs to request-sending instructions. The paper evaluates Sheriff on the Tranco top 10K sites and constructs proof-of-concept exploits across several sites. Finally, the paper evaluates the adoption and efficacy of existing countermeasures against client-side request hijacking attacks.

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) The Sheriff tool presented in this paper extends and improves upon existing tools, and will be made publicly available to enable future research.
2) The review of browser API capabilities and Web specifications, and the systematization of request hijacking vulnerabilities provides a valuable framework for reasoning about such vulnerabilities, and sheds light on vulnerabilities that have been overlooked in the past. It also supports the development of tools like Sheriff.
3) The analysis of the Tranco top 10K sites provides useful insights into the prevalence of request hijacking vulnerabilities. The review of the adoption and efficacy of existing countermeasures motivates the need for further work in this area.