Do (Not) Follow the White Rabbit: Challenging the Myth of Harmless Open Redirection

Soheil Khodayari†, Kai Glauber*, and Giancarlo Pellegrino†
†CISPA Helmholtz Center for Information Security, *Saarland University
{soheil.khodayari, pellegrino}@cispa.de, s9kaglau@stud.uni-saarland.de

***Abstract*—Open redirects are one of the oldest threats to web applications, allowing attackers to reroute users to malicious websites by exploiting a web application's redirection mechanism. The recent shift towards client-side task offloading has introduced JavaScript-based redirections, formerly handled server-side, thereby posing additional security risks to open redirections. In this paper, we re-assess the significance of open redirect vulnerabilities by focusing on client-side redirections, which despite their importance, have been largely understudied by the community due to open redirect's long-standing low impact. To address this gap, we introduce a lightweight, static-dynamic system, STORK, that detects open redirect vulnerabilities by extracting and using vulnerability indicators, which is designed with scalability and cost reduction objectives. Applying STORK to the Tranco top 10K sites, we conduct a large-scale measurement, uncovering 20.8K open redirect vulnerabilities across 623 sites and compiling a catalog of 184 vulnerability indicators. Afterwards, we use our indicators to mine vulnerabilities from snapshots of live webpages, Google search and Internet Archive, identifying additionally 326 vulnerable sites, including Google WebLight and DoubleClick.**

Then, we explore the extent to which their exploitation can lead to more critical threats, quantifying the impact of client-side open redirections in the wild. Our study finds that over 11.5% of the open redirect vulnerabilities across 38% of the affected sites could be escalated to XSS, CSRF and information leakage, including popular sites like Adobe, WebNovel, TP-Link, and UDN, which is alarming. Finally, we review and evaluate the adoption of mitigation techniques against open redirections.

I. Introduction

HTTP redirections are commonly used to guide users from one resource to another. While traditionally employed by server-side programs to signal the new or temporary locations of web resources (i.e., 3xx HTTP responses [1]), web applications nowadays also utilize them within client-side JavaScript programs, supporting functionalities such as redirecting to a landing page after a successful login and navigating to user-specific dashboards. Often, the target destination is specified through a URL parameter, which the web application employs to direct users. However, when this parameter is not adequately validated, the web application becomes susceptible to an *open redirection* vulnerability.

Open redirect vulnerabilities have been somewhat understudied by the research community, which has primarily focused on their detection via *indicators*, searching for destination URLs in the query strings of links, i.e., [2, 3]. The lack of interest in these vulnerabilities could be attributed to the relatively low prevalence, only making up 1% of the 237,470 CVE entries, compared to Cross-Site Scripting's 37%[1], and limited exploitation scenario, where attackers use vulnerable sites to mask malicious URLs [5, 6], like phishing links [7–9], without directly harming the vulnerable site itself. Vulnerability disclosure programs, including reputable ones like Google [10] and Microsoft [11], often do not consider reports of open redirects as qualifying issues eligible for rewards. In rare instances, attackers can leverage open redirects to escalate to more severe threats, such as XSS via `javascript` URIs [12–14] or request forgery [15–17].

This paper re-evaluates the long-standing low security risk of open redirections by focusing on client-side open redirects at scale, exploring the extent to which their exploitation can lead to more significant threats. The cornerstone of our study is the *detection* of open redirects on websites in a lightweight manner, reducing the cost to detect significant and impactful vulnerabilities arising from client-side open redirect variants. While client-side static analysis has proven valuable in numerous studies of client-side vulnerabilities, such as client-side Cross-Site Request Forgery (CSRF) [16, 18] and DOM clobbering [19], employing static analysis techniques to process hundreds of thousands of pages is generally resource-intensive, resorting to sampling strategies as a compromise between the breadth of coverage and the feasibility of the study within a reasonable timeframe. For instance, previous work [16] analyzed 39% of the 867K collected webpages, underscoring the limitations of relying solely on static analysis for large-scale studies. Previous works have demonstrated that manually-curated indicators could serve as a cost-effective method for identifying open redirection vulnerabilities [2, 3]; however, achieving a comprehensive list of indicators through manual analysis is challenging in practice.

Our Approach. In light of this, we propose a novel cost-reduction methodology, named STORK, that combines both ideas, offering a lightweight detection trade-off compared to the costly static analysis. First, we use static analysis on a subset of pages to find client-side open redirects, and confirm the vulnerabilities with test payloads dynamically. From the

[1]We conducted a case-insensitive keyword search of `xss` and `open redir` in the CVE database [4].

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240523
www.ndss-symposium.org

confirmed cases, we extract indicators that we use as search keywords to find other candidate webpages from the larger dataset, which we confirm with test payloads. Mining with indicators offers remarkable flexibility, enabling us to expand our search to additional datasets and open redirect variants, such as Google search via Google dorking [20–22] or Internet Archive [23], and detecting both client-side and server-side open redirections.

Starting from the 867K pages collected in [16], we define and execute search queries on a subset (339K pages) to identify open redirect vulnerabilities by constructing and traversing JavaScript property graphs [18]. We derived indicators by grouping vulnerable URLs based on similarities, considering features like syntax and injection points (e.g., path or query parameter). We found 20.8K confirmed open redirections across 623 websites, and extracted 184 indicators. We then use the identified indicators to search for matching URLs in the remaining 528K pages, in Google Search, and in the Internet Archive considering the top 10K domains. This way, indicators narrowed down the test set from about 4M pages we collected to only 214K, from which we confirmed 375 ulnerabilities via dynamic testing, including popular sites like Google WebLight, Starz and DoubleClick. Our study reveals that open redirect vulnerabilities are widespread, impacting ∼8.7% of the top 10K websites.

Then, we conducted a comprehensive exploitability and threat escalation analysis using both automatic and manual testing, covering DOM-based XSS [24, 25], client-side CSRF [18, 26], and information leakage [17], revealing that client-side open redirects could have broader implications. Particularly, we constructed proof-of-concept exploitations for 332 sites, including popular sites like Adobe, WebNovel, TP-Link, UDN, Lexmark, and VK. Overall, our results illustrates a concerning landscape, where about 11.5% of the open redirect vulnerabilities could be escalated to more critical threats.

Finally, we examined the array of mitigation strategies utilized by websites with closed redirections. Through semi-automated analysis of 4K sites, we identified six distinct types of mitigation techniques, with redirect notice pages, input validation, and Content Security Policy [27] being the most widely adopted countermeasures.

Insights. Our comparison of indicator-based vulnerability detection with static analysis suggests that it is about 100 times faster and uses 13 times less storage than static analysis, making it highly scalable. However, we found that indicators may result in more false negatives, which is influenced by how effectively crawlers capture various URL parameters linked to different code execution paths, since indicators operate at the URL level. For instance, when using the JAW crawler [18], we observed a 76% false negative rate of open redirects. Although static analysis detects more open redirect vulnerabilities, indicator-based findings have a higher rate of XSS escalations (22% vs. 8%). Furthermore, we found that indicators can identify vulnerabilities static analysis misses, which is primarily caused by limitations of static analysis (e.g., handling dynamic features like reflection). Overall, our results show that indicators could serve as a valuable trade-off and enable us to cast a wider net.

Listing 1: A simplified client-side open redirect vulnerability derived from `lexmark.com`.

```
function printView(url){
  if (url.indexOf('lexmark.com') > 0){
    let loc = window.location
    let sep = (loc.search === "")? "?": "&"
    let query = loc.search + sep + "view=print"
    loc = url + query
    window.location.replace(loc)}
} // [...]
window.addEventListener('hashchange', (e) => {
  var h = window.location.hash.slice(1)
  if(h.indexOf("print;") > 0){
     var url = h.split(";")[1]
     printView(url)
  }});
```

In summary, this paper makes the following contributions:

- We present STORK, a cost-reduction method to detect open redirects by extracting and using vulnerability indicators, uncovering 20.8K vulnerabilities across 623 sites, and a catalog of 184 indicators divided in nine groups.
- We use our indicators to mine vulnerabilities from top 10K live websites, Google search and Internet Archive, identifying 375 additional vulnerabilities in 326 sites, highlighting the potential of our indicators for vulnerability discovery.
- We quantify the impact of open redirections in the wild, showing that over 11.5% of the vulnerabilities across 38% of the affected sites could be escalated to XSS, client-side CSRF and information leakage.
- We review and evaluate the adoption of open redirect mitigations in the wild, identifying redirect notice pages, input validation, and CSP as the most common countermeasures.

II. Background

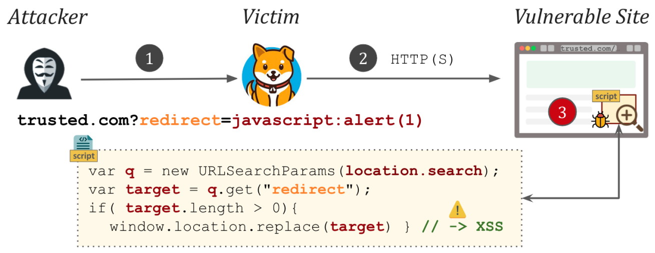
Before presenting our study, we first introduce and dissect open redirect vulnerabilities (§II-A), and then, we present the threat model of this work (§II-B).

A. Open Redirect Vulnerability

Open redirect vulnerabilities [2, 3, 28, 29] originate when web applications use untrusted inputs in HTTP requests (e.g., URL query parameters' values) to forward users to a destination resource. If such request parameters are not (properly) validated, attackers can redirect users to arbitrary external Web resources, such as phishing, malware, and other malicious content [2, 3, 7, 28–31]. Both client-side and server-side programs can perform redirections. For example, server-side code can use the `Location` [32] or `Refresh` [33] HTTP response headers to trigger an HTTP redirect. Client-side redirects, however, occur via JavaScript code or the HTML `meta` tag.

Listing 1 shows a real snippet of vulnerable client-side code (disclosed and patched), which uses URL hash fragments to

Fig. 1: Example exploitation of a client-side open redirect vulnerability.



```
trusted.com?redirect=javascript:alert(1)
```

```
var q = new URLSearchParams(location.search);
var target = q.get("redirect");
if( target.length > 0){
  window.location.replace(target) } // -> XSS
```

open a destination webpage suitable for printing. In more details, the code first listens for changes in the URL fragment through an event listener (line 9). Whenever the fragment changes, it checks if it contains the constant string `"print;"` (line 11), retrieves the string after it in the fragment (line 12), and calls the function `printView()` by passing this value (line 13). The function `printView()` accepts a protocol-relative URL, attempts to check if the URL belongs to a trusted domain (line 2), modifies it by appending the query parameter `view=print`, and redirect the current page to the resulting value, i.e., the variable `loc` (lines 3-7). The vulnerability originates in the assignment in line 7 because attackers can control the value of variable `loc` through the webpage URL hash fragment, and ultimately pick the destination webpage of their choosing because the code does not correctly validate the URL string passed as input to the `printView()` function but use it as a part of the destination of the redirect, e.g., attackers may bypass the validation check in line 2 with a URL payload like `lexmark.com.evil.com`.

B. Threat Model

In this paper, we consider a regular *web* attacker [34, 35] who can exploit open redirect vulnerabilities by injecting attack payloads containing malicious URLs as the destination target of HTTP redirections in trusted URLs, and lure victims into visiting them, which is in line with prior research [2, 3, 18, 19, 24, 36, 37]. The injection can happen by manipulating various JavaScript input sources, including the URL, window name, document referrer, and postMessages.

There are two attack models depending on the input source. First, a web attacker can craft a malicious URL, belonging to the origin of the honest but vulnerable web site, that when visited by a victim leads to a redirection to an attacker-controlled domain. Alternatively, for window name, document referrer, and postMessages, a web attacker can control a malicious page and use browser APIs to trick the vulnerable JavaScript of the target page to cause the HTTP redirection.

Attacks. Open redirect vulnerabilities are commonly abused as a part of social engineering attacks, such as phishing. However, as we will show in §VI, the risk and impact may extend further in the context of client-side code vulnerabilities. In this paper, we study to what extent we can escalate client-side open redirect vulnerabilities, focusing on three critical classes of Web attacks: Cross-Site Scripting (XSS), request forgery and sensitive information leakage.

Firstly, client-side open redirections can be escalated to arbitrary code execution if the attacker can choose the `javascript` scheme as the destination target of the redirect. For instance, as illustrated in Listing 1, an attacker could achieve XSS by choosing the attack payload `javascript:alert(document.cookie+"lexmark.com")`. Figure 1 demonstrates this attack scenario. Furthermore, if the redirection URL contains sensitive information such as authorization codes or OAuth tokens [15, 38], attackers may exploit the redirection chain to steal such data. Finally, recent studies (e.g., [39, 40]) have demonstrated that applications may employ GET requests to implement state-changing operations (e.g., deleting an entry from database). This enables attackers to abuse client-side open redirect vulnerabilities to generate arbitrary forged requests to state-changing endpoints, achieving client-side CSRF [18, 26]. In comparison, server-side redirects can be abused for phishing [2, 28, 29] and SSRF attacks [41, 42]. In this paper, however, we focus on escalations of client-side open redirects, which has been largely dismissed by prior work [2, 3].

C. Open Redirect vs. Request Hijacking

As discussed in §II-A, open redirect vulnerabilities can impact both server-side and client-side programs. Client-side open redirects are a specific instance of request hijacking vulnerabilities [16]. Request hijacking occurs when an attacker manipulates inputs to request-sending APIs, such as the request URL and body. When these manipulated inputs lead to a top-level navigation to a different domain, it constitutes a client-side open redirect. Although recent research [16] have explored request hijacking, they have not covered the detection of server-side open redirects and their defenses, nor extensively analyzed the exploitability of forgeable, top-level client-side requests for open redirections, particularly on a large scale. This paper extends the existing knowledge by studying both client-side and server-side open redirect variants, demonstrating that the indicators we identify for client-side open redirects are not only effective but also applicable to server-side redirections, providing a more comprehensive understanding of open redirects across different programming contexts.

III. Overview

This section provides an overview of our methodology (§III-A) and a brief description of our approach (§III-B).

A. Methodology

Step 1–Vulnerability Detection and Indicators. The first part of our paper studies the correlation between URL structure and open redirect vulnerabilities in real websites, extracting various patterns that could indicate the presence of vulnerabilities. In particular, we focus on two main aspects: (i) building a lightweight framework to characterize client-side open redirect vulnerability patterns leveraging static and dynamic analysis, and instantiating the framework against in-the-wild websites to create a catalog of these indicative patterns; (ii) reviewing existing vulnerability reports and CVEs to identify past instances of open redirects and their patterns, enriching our list of indicators also with patterns of *server-side* open redirects. We show that a significant fraction of the open redirect vulnerabilities converge toward a few distinct patterns. We address this step in §IV.

Step 2–Vulnerability Mining and Prevalence. After creating a comprehensive database of vulnerability indicator patterns, we use them to extract potential candidates of new vulnerabilities from public data archive repositories (i.e., Internet Archive and Google Search). Then, we use dynamic analysis to confirm the presence of the vulnerability, uncovering the potential of vulnerability mining for scaling up and detecting new vulnerabilities in a lightweight manner. Finally, we quantify the prevalence of open redirects in the wild using snapshots of live websites leveraging indicator-based vulnerability mining.

Step 3–Exploit Analysis and Escalation. After mining open redirects from public data, we study the impact and severity of the discovered vulnerabilities. In particular, we study the variety of threats that arise from open redirects and explore to what extent we can escalate them to more severe attacks like cross-site scripting, request forgery, and sensitive information leakage. While open redirects have been commonly abused as a part of social engineering attacks (e.g., phishing), we show that the risk and impact could go further as they could be directly exploitable themselves. We present this step in §VI.

B. Our Approach: STORK

In this section, we present an overview of the design and implementation of STORK, a framework to study open redirect vulnerabilities at scale, providing a fast and cost-effective trade-off to pure static analysis. Figure 2 presents an overview of our approach. Broadly, it has three main components corresponding to each of the steps outlined in §III-A: ❶: automatic extraction of vulnerability indicators, ❷: vulnerability mining using indicators, and ❸: exploitation analysis performing run-time tests for attacks.

Given a list of sites as input, STORK can collect the snapshots of their webpages, or reuse existing snapshots. Then, it performs static analysis by constructing a property graph model, tracing data flows from program inputs to JavaScript instructions that trigger a redirection. Afterwards, it confirms the presence of an open redirect vulnerability by conducting run-time monitoring tests, and employs confirmed vulnerabilities to extract indicators by grouping vulnerable URLs together based on their similarity. With a catalog of indicators at its disposal, STORK mines potential vulnerabilities from snapshots of webpages, such as those in the Wayback machine or Google archived pages, and verifies the presence of open redirection via dynamic testing. Lastly, we examine the potential escalation of the open redirect vulnerability to more critical vulnerability classes either automatically or manually, e.g., DOM-based XSS by dynamically testing a dictionary of benign attack payloads.

IV. Vulnerability Indicators

The first part of this paper intends to extract vulnerability indicators by detecting and studying real open redirect vulnerabilities in web applications. We first present our vulnerability detection pipeline (§IV-A), and then describe how we instantiated it at scale to identify vulnerability indicator patterns in the wild (§IV-B), i.e., step ❶ of Figure 2.

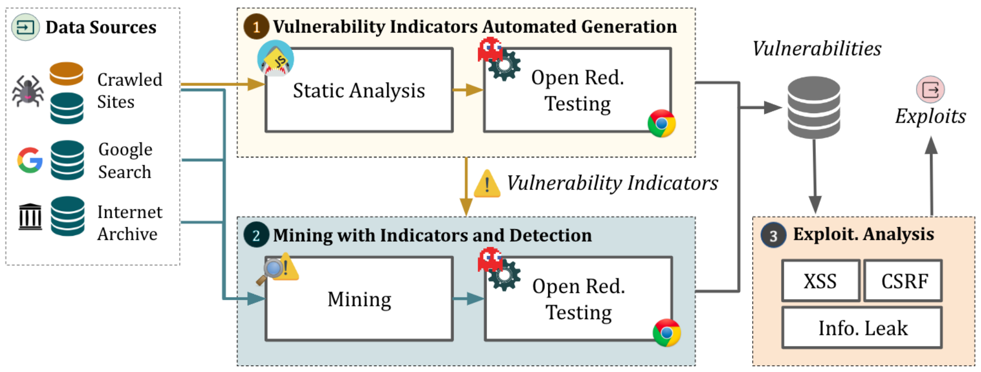
A. Vulnerability Detection

Starting from a website, STORK creates a graph-based model of the program and use it to perform static analysis, finding unvalidated data flows from JavaScript program inputs to instructions that trigger a redirection. Given the data flow and the webpage URL, it generates a set of candidate test cases. Finally, it executes the test case dynamically to see whether it can observe the client-side redirect at runtime, eliminating potential false positives. The details follow.

IV-A1 Data Collection. STORK can gather the client-side code of web applications for security testing. In this study, we reused the crawler and dataset (i.e., snapshots of webpages) provided by prior work [16]. The crawler is based on Playwright [43] and an instrumented version of Firefox (v98.0.2) [44, 45]. When provided with a domain as input, it employs a depth-first strategy to navigate webpages, halting its exploration either when no new URLs are discovered or after visiting a maximum of 200 URLs per site. Throughout this exploration, it gathers webpage resources (e.g., scripts), DOM snapshots, and network messages [18, 19].

IV-A2 Static Analysis. After gathering the client-side code of web applications, we model them as a Code Property Graph (CPG)[18, 46]. CPGs are graph-based representations of a program, amalgamating various code representations to capture both syntactical and semantical aspects. CPGs unify different models, including the Abstract Syntax Tree (AST), Control Flow Graph (CFG), Call Graph (CG), Program Dependence Graph (PDG), and Event Registration, Dispatch, and Dependency Graph (ERDDG). These models depict the hierarchical structure of a program's syntax, the order and conditions governing the execution of program instructions, the function call relationships, the data flow and control dependencies within program statements, and event-driven control transfers, respectively. In this paper, we extend and use the static analysis engine of JAW [18] to create a CPG for each webpage. We import each CPG into a Neo4j [47] graph database, which we can query using the Cypher language [48] for security testing. We made several enhancements to JAW for improved control and data flow analysis capabilities. For example, we added support for arrow function expressions [49] and asynchronous `setInterval()` calls [50], improving the precision of PDG edges and call graph. Additionally, we introduced handling for the `globalThis` object [51] to improve pointer analysis operations. Furthermore, we added support for promise-based callbacks via methods like `Promise.then()` [52], which improves control transfer modeling and def-use analysis tasks [53]. Finally, to improve scalability, we implemented

Fig. 2: Overview of STORK. The figure shows the pipeline to (i) detect open redirect vulnerabilities via program analysis and extract vulnerability indicators, (ii) using fast indicator searches for vulnerabilities, and (iii) analysis of the exploitability of the confirmed open redirections for escalation to critical attacks.



several optimizations by migrating part of the call graph generation (e.g., resolving aliased pointers) to C++. Overall, these modifications addressed several of the shortcomings of JAW, enabling more precise analysis and improved scalability in the construction of CPGs.

Then, we frame the task of open redirect vulnerability detection as a graph traversal problem on CPGs, where we intend to trace data flows originating from attacker-controllable inputs, such as URL parameters, hereafter sources, to JavaScript instructions that cause a redirection or navigation event, hereafter sinks. We extracted the list of sources/sinks by reviewing the Web API specifications [54], including all sources and redirection sinks in related work [16, 18, 24, 36, 45, 55] and testing tools [44, 56–59], resulting in a comprehensive list (Table II). To accomplish this task, we created a series of queries to identify each source and sink in the CPG. Afterwards, we conduct backward data flow analysis from sinks to sources (i.e., program slicing [60]), determining whether a source value reaches a sink instruction. This component outputs the set of potential data flows found and the injection point for each flow, which we verify via dynamic analysis, as discussed next.

IV-A3 Test Generation and Attack Techniques. Given a potential data flow, the goal of this step is to prepare candidate test URLs or test webpages (e.g., for postMessage-based redirects) for dynamic vulnerability confirmation. Note that in case of test webpages, they open the target webpage via `window.open()` API [61]. Therefore, we can use browser APIs to insert the input, e.g., set the name of the opened window via `window.name` API [62] or send postMessages to it [55, 63].

To do this, STORK inserts test payloads in the injection point. In particular, STORK uses a pre-defined list of payloads that we manually compiled, covering a comprehensive array of attack techniques. We systematically reviewed academic literature [2, 64–67], HackerOne vulnerability reports [68], the CVE database [4, 69], Stack Exchange [70] and Dev [71] security communities, and other non-academic resources (see, i.e., [7, 15, 28–31, 33, 72–77]), looking for open redirect attack payloads and general URL filter bypass techniques. We consider in scope those techniques that can be exploited by a

	Total	Unique	P_1	P_2
Webpages	1,034,521	867,455	339,267	528,188
Scripts	46.1 M	36.7 M	11.5 M	25.2 M
Lines of Code	129.8 B	104.1 B	32.4 B	71.7 B

Legend: P_i= Portion i .

TABLE I: Statistics of the dataset for the top 10K sites.

standard web attacker [34, 35]. In total, our review identified 26 distinct techniques, which we further grouped into eight categories based on their similarity (e.g., the component they target like URL path vs. scheme, or the type of the operation). Table X of Appendix A summarizes our findings. For each potential data flow, STORK generates test URLs or webpages with a payload from each of the 26 techniques in Table X.

IV-A4 Dynamic Verification. Given a set of test URLs or webpages associated with each dataflow, this component examines them to confirm the presence of open redirect vulnerabilities. Specifically, the test payloads contain the address of a local server that the verifier controls and a unique ID. To execute the tests, the verifier visits the test URL or page in a browser, and subsequently checks whether it receives a request on the local server with a matching ID and if the target webpage frame redirected to it. If these conditions are met, the data flow is flagged as an open redirect vulnerability. Conversely, if all the generated tests for a specific data flow fail, the verifier dismisses it as a false positive.

B. Vulnerability Indicators and Prevalence

In this section, we conduct the largest-to-date study to detect open redirect vulnerabilities in the wild, with the overarching goal to identify and extract indicator patterns from real vulnerabilities. The rest of this section details statistics about the dataset and the analysis steps.

IV-B1 Data Collection and Processing. In this paper, we reused the snapshots of webpages provided by prior work [16], which is based on the Tranco site list downloaded on Sept. 29, 2022 (ID: N7QWW) [78], and collected in Oct. 2022 during a six week period. The dataset contains a total of 1,034,521$\sim$1M webpages across top 10K sites. These 1M pages contained around 46.1M scripts with over 129.8B LoC. Page de-duplication 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. We divided this dataset in two portions, one for extracting patterns of vulnerabilities, and the other one for searching the presence of the vulnerable patterns, which we call P_1 and P_2, respectively. The first portion P_1 contains a maximum of 50 pages per site that have the highest frequency of dynamic data flows [44], resulting in a dataset of 339,267 webpages with 32.4B LoC, which is similarly to prior work [16]. The second portion P_2 contains 528,188 pages with 71.7B LoC. Table I summarizes the dataset statistics.

IV-B2 Program Analysis. Given as input the P_1 dataset, STORK performs static analysis for vulnerability discovery. We processed an average of 34 scripts and 95K LoC per page, generating 339K HPGs. Afterwards, STORK performed graph traversals to detect data flows from JavaScript program inputs to redirection instructions. In summary, STORK identified an average of five redirection sinks and 65 sources per webpage, totaling about 22.3M sources and 1.7M sinks. Among these, static analysis found a total of 25,990 potential data flows from sources to sinks, of which about 80% (i.e., 20,898) have been confirmed via dynamic testing. In summary, these vulnerabilities affected 11,155 webpages across 623 websites, of which 20,471 flows across 599 sites originate from URL-based sources (e.g., query parameters), whereas 427 flows across 39 sites are from non-URL sources (e.g., postMessages). Table II summarizes the results.

IV-B3 Analysis of Vulnerabilities. We found that a small fraction of vulnerable redirections (i.e., 427 or about 2% of cases) originate from sources other than webpages' URL parameters, which cannot be detected by traditional detection approaches based on URL parameter fuzzing (e.g., [2, 3]). In comparison, STORK's SAST component can detect them, and STORK's DAST component can verify them. In contrast, about 98% of open redirections originate from URL sources. As we will show next in §IV-B4, these vulnerabilities can be largely detected by pattern-based searching provided that a comprehensive list of indicative patterns is available. For example, Table IX (appendix) presents the top 10 URL query parameter keys featuring the highest number of distinct domains utilizing the parameter for (open) redirections. We observed that the most prevalent parameter in open redirects is `url` which is used by 102 domains across 1,224 unique URLs, followed by `domain` and `redir` keys across 52 and 39 domains, respectively. Consequently, these indicators can be leveraged to search for potential open redirects, reducing the overall effort for program analysis.

IV-B4 Pattern Extraction and Indicators. After identifying open redirects, we group them together to extract common patterns. In addition to the vulnerabilities we discovered in this section, we manually analyze existing vulnerability reports from the MITRE CVE database [4, 69], which we collected in §IV-A3, and extract the affected URLs from each report. This can augment our dataset with information about **server-side** open redirects. In total, we identified 687 CVEs for open redirect vulnerabilities, of which in only 460 cases, we were able to retrieve the affected endpoint, either directly from the report or by following the links provided.

To extract vulnerability indicators, we grouped the URLs together based on their similarity by abstracting away the specific domain affected, and considering the syntax, injection point and the position (e.g., path or query parameter), and the values of the redirection parameters, decomposing URLs to their building block components.

Starting from the 20,471 confirmed URL-based open redirections we discovered, and the 460 past CVEs of open redirections, we extracted a total of 184 concrete vulnerability patterns, of which 95 are new (i.e., discovered exclusively using our dataset). We grouped these 184 concrete patterns into nine distinct categories by abstracting away the specific redirection parameter in the URL (e.g., "next" vs. "redirect"). Our results show that out of these nine indicator patterns, three are new (as we found no existing CVEs revealing similar structural pattern), two include new variants (as the general structure is the same, but the specific redirection parameter in the URL is different), and the remaining four are similarly to the known cases (i.e., both the general structure and redirection parameters match to known cases). We further grouped these nine patterns into three different classes based on the position of the redirection parameter in the URL (i.e., query parameter, path or fragment). Table III summarizes the results.

IV-B5 Analysis of Indicators. Unsurprisingly, we observed that a significant fraction of the vulnerabilities occur when using query parameters for redirections. Particularly, open redirects via pattern A1 are the most widespread, being present on more than 14,201 vulnerabilities across 402 sites and 382 existing CVEs. In comparison, vulnerabilities relying on the URL path segments for redirections demonstrated a moderate level of presence, with the most popular being pattern B1 representing 948 vulnerabilities across 147 sites and 35 CVEs. We observed that a significant fraction of the open redirect vulnerabilities affecting client-side code (i.e., 12.4%) rely on hash fragments. The widespread usage of URL parameters for redirections, coupled with a wide variety of potential vulnerabilities, presents a tantalizing attack surface for hackers. The remainder of this paper is dedicated to using these indicators for mining vulnerabilities from snapshots of webpages.

IV-B6 Coverage of Indicators. We found that the CDF tracking the growth of indicator patterns across randomly-ordered vulnerable webpages in P_1 dataset reaches saturation at about 66%, suggesting that our patterns are comprehensive (see Figure 3 of Appendix A). Furthermore, as we will show in §V-C, our baseline experiments did not reveal false negatives due to missing patterns, further reassuring comprehensiveness. However, we do not claim nor guarantee that our patterns are exhaustive, as they are influenced by crawling coverage (e.g., deep and authenticated states) and limitations of static analysis (e.g., handling dynamic JavaScript features).

Sink / Source	loc.href	loc.hash	loc.search	doc.uri	Flows	Verified	win.name	doc.ref	pMsg	Flows	Verified	Total	Verified	Pages	Sites
win.open()	17,321	2,078	1,120	55	20,574	16,480	21	76	3	100	86	20,674	16,566	8,846	455
win.loc	862	53	42	10	967	743	12	16	1	29	10	996	753	543	168
frame.src	41	261	7	0	308	202	16	8	35	59	51	367	253	202	29
loc.href	2,654	178	186	1	3,019	2,597	34	89	0	123	78	3,142	2,675	1,627	302
loc.replace()	381	12	8	0	401	358	3	0	7	10	2	411	360	281	62
loc.assign()	70	5	72	0	147	91	251	3	0	254	200	401	291	184	13
Total	21,329	2,587	1,434	66	25,415	**20,471**	337	192	46	575	427	25,990	**20,898**	11,155	**623**

TABLE II: Summary of the vulnerable data flows found by STORK. Columns represent different client-side sources. The left part shows flows originating from URL parameters, while the middle part shows flows originating from other sources like window name and postMessages (i.e., pMsg column).

V. Vulnerability Mining

In this section, we leverage the vulnerability indicators we discovered in §IV to mine potential vulnerabilities from public data archive repositories in a cost-effective way, as shown in step ❷ of Figure 2. Our primary focus is on data platforms such as Google Search and Internet Archive. Additionally, we demonstrate the versatility of our technique by applying it on live websites (i.e., the second portion of the dataset), quantifying the prevalence of open redirect vulnerabilities in the wild. Finally, we study the cost-benefit trade-offs of indicator-based and static analysis-based vulnerability detection, showing that indicator-based scanning is up to 100x faster, and can detect vulnerabilities static analysis misses, but may also introduce more false negatives. Our results emphasize the effectiveness of indicator-based vulnerability scanning as a lightweight method for detecting open redirects at scale, including both client-side and server-side variants.

A. Indicator Mining

Having compiled an extensive catalog of vulnerability indicator patterns, we now employ these patterns to identify potential candidates of zero-day open redirect vulnerabilities. We searched the Internet Archive and Google page snapshots for our indicator patterns for pages archived within the past two months (June 2023 - July 2023), and use the resulting URLs as candidates for security testing. We used archives to collect a large pool of URLs (not webpages), on which we searched our indicators, and conducted our experiments exclusively on live webpages of archive URLs.

V-A1 Data Collection and Mining. For each vulnerability indicator pattern, we created a corresponding regular expression and a Google dork query [21, 22]. First, for each of the top 10K websites, we applied the regular expressions to search for matching patterns on Internet Archive and (snapshots of) live websites (i.e., P_2 in Table I). For Internet Archive, we relied on the CDX server API [23] for our search. Second, for each website, we use the Google Custom Search JSON API [20] to look for the dorks that we created. The custom search API, limited to returning a maximum of 10 results per query request, guided our study with the understanding that an attacker would aim for cost-effectiveness when exploiting this method, aligning with our threat model of §II. At the time of writing this paper, Google's Custom Search JSON API provides 100 search queries per day for free, and charges about 5 USD per 1000 queries [79]. Accordingly, we opted to submit only one request per Dork query and site, enabling us to utilize the free tier and minimize additional costs, totaling no more than 50 USD for 10K sites.

V-A2 Overview of Query Results. Table IV shows the number of candidate URLs found for open redirects within the top 10K sites across the three considered data sources. In total, our queries identified about 4M candidate URLs, the majority of which belonged to Internet Archive. URL de-duplication in each data source enabled us to reduce the number of candidate URLs substantially to as few as 215K cases (see Appendix D). However, we observed that a small fraction (i.e., 0.5%) of the unique URLs from different data sources are also duplicates of one another, further reducing results to 214K unique cases. Table XI (Appendix A) presents the distribution of these URLs across different vulnerability patterns.

B. Vulnerability Verification

After identifying candidate URLs for open redirects, we perform run-time tests to eliminate false positives, as shown in step ❷ of Figure 2. We follow a similar approach presented in §IV-A3. Specifically, for each URL, we test one payload for each of the 26 attack techniques in Table X, i.e., we put the payload in the injection point according to the vulnerability patterns of Table III, and subsequently examine whether it causes the page to redirect to our web server. As soon as a payload goes through in one of the tests, we mark the endpoint as vulnerable. If all of the test cases fail, we exclude the candidate URL from the output.

V-B1 Overview of Vulnerabilities. In total, our vulnerability mining approach combined with dynamic testing uncovered 375 new open redirect vulnerabilities across 326 websites, of which the majority (i.e., 70% or 265) were discovered using Internet Archive URLs. Upon examining vulnerabilities identified through simultaneous analysis of multiple data sources, we found six instances where a vulnerable URL occurred in both the Internet Archive and our live website crawl. We refer interested readers to Table XI of Appendix A.

V-B2 Analysis of Type of Redirects. Manual analysis of the 375 vulnerabilities indicated that 204 cases involved client-side redirects, of which the overwhelming majority (i.e., 202) were JavaScript-based, and the remaining two employed `meta` tags for redirection. In contrast, we observed that 171 vulnerabilities occurred due to server-side redirections. This finding

Type		ID	Pattern	Params	Count	New	Example	CVEs	Vulns	Sites
Query	⊞	A1	`?P=CONST`	R1	109	59	?next=example.com	382	14,201	402
		A2	`?CONST=https%3A%2F%2F | www. | DOMAIN.PSL`	-	3	0	?xyz=https%3A%2F%2Fexample.com	12	2,360	91
Path	⊞	B1	`/P/https%3A | DOMAIN.PSL`	R2	17	1	/callbackUri/www.example.com%2Findex	35	948	147
		B2	`[/CONST]/https%3A/P`	R3	13	0	/https%3A%2F%2Fexample.com/submitUrl	23	260	24
		B3	`/CONST/https%3A | DOMAIN.PSL`	-	2	0	/index.php/example.com%2Findex	2	122	7
		B4	`/https%3A/CONST/`	-	1	0	/https%3A%2F%2Fexamle.com%2Findex/get	6	31	3
Hash	◼	C1	`#P=CONST`	R4	35	35	#ajaxUI=example.com/profile/index	0	2,207	108
	◼	C2	`#CONST=https:// | DOMAIN.PSL`	-	2	2	#u=https://example.com	0	311	26
	◼	C3	`#https:// | DOMAIN.PSL`	-	2	2	#example.com/profile/index	0	31	2
Total					**184**	**95**		460	**20,471**	**599**

TABLE III: Open redirect indicator patterns for vulnerability mining, grouped by the URL segment responsible for the redirection. The table shows the number of vulnerabilities matching each pattern. Rows marked with ◼ represent new patterns, whereas ⊞ marks variants where a known pattern is observed with a new parameter as in Table VIII. Legend: `P`= values in "params" column; R_i= row i in Table VIII; CONST= constant string $\neq$ `P`; []= optional part; |= OR operator.

Source	URLs	Unique
S1: Internet Archive	4,001,896	188,403
S2: Google Cache	2,313	1,237
S3: Live Crawl	29,294	26,163
Total	4,033,503	214,645

TABLE IV: Candidate URLs found for open redirects in top 10K sites.

is not surprising, because the study of open redirect CVEs in §IV-B4 showed that our automatically-generated catalog of client-side indicators are a superset of the server-side variants. In line with this finding, experimental results demonstrate that our indicators can capture both client-side and server-side open redirect variants.

V-B3 Precison of Vulnerability Indicators. We observed that about ∼4% of the sites that matched our indicator mining queries (Cf. Table III) were open redirect vulnerabilities. When looking at candidate URLs, almost 2 of every 1K candidate URLs was an open redirect, which increased to a rate of up to 18 per 1K URLs for the Google search API. Therefore, only a small fraction of cases matching indicators represent actual vulnerabilities. However, indicators reduced the search space significantly. For example, for live sites (P_2 dataset in Table I), indicators quickly narrowed the testing scope from 528K webpages to about 26K candidates, resulting in a significant optimization factor of ∼20 for dynamic testing.

In general, precision of indicator patterns themselves is not a major concern, because our dynamic indicator-based vulnerability scanning approach does not produce any false positives. Specifically, following the approach described in Sections IV-A4 and V-B, STORK verifies potential open redirect vulnerabilities by conducting runtime tests using the payloads enumerated in Table X. STORK uses a Playwright-controlled browser [43] to visit a target URL containing a test payload, and flags it as an open redirect vulnerability only when it detects that the target webpage redirected to an arbitrary, controlled page at runtime. To demonstrate that this approach is robust to false positives, we manually analyzed all the 375 vulnerabilities identified in §V-B1. In particular, we manually loaded the URL containing the found attack payload via the automated approach in the browser, and checked whether the redirection happens and is open to arbitrary destinations. The results confirmed that there are no false positives. This finding was expected, as contrary to static analysis, dynamic analysis techniques (e.g., [2, 3, 57, 58]) typically produce little-to-no false positives.

C. Cost-Benefit Analysis

The main contribution of indicators is enabling larger-scale analyses compared to costly static analyses. However, indicator-based, dynamic vulnerability scanning may also result in false negatives. In this section, we evaluate and compare the cost-benefit trade-offs between using indicator-based vulnerability mining and static analysis methods for identifying open redirects. Our methodology is as follows. We chose 50 applications at random from the P_2 dataset in Table I, encompassing a total of 42,288 webpages (hereafter P_2'), and compare the analysis and verification time, storage requirements, and performance of each approach.

V-C1 Performance. After analyzing 42,288 URLs, static analysis identified 58 potentially vulnerable data flows, with 46 of these dynamically confirmed as open redirects, translating to a false positive rate of about 20%. These vulnerabilities impacted 46 pages across **eight** applications. In comparison, indicator-based vulnerability scanning immediately narrowed the scope to 3,011 candidate URLs for testing, and found sixteen cases as open redirects across **six** applications. Notably, five out of these sixteen vulnerabilities were exclusively found by indicators, not detected by static analysis due to the absence of call and PDG edges in CPGs, and because one of the five vulnerabilities was a server-side open redirect. This shows an important advantage of indicators—they can uncover security flaws that client-side static analysis might miss. When looking at individual vulnerabilities, indicators also showed high false negatives compared to static analysis (i.e., 76%). We found that these FNs arise because indicators operate at URL level and their optional parameters (e.g., query and hash) are missing from the URLs collected by the crawler or archives, which can trigger different code execution paths. However, static analysis can capture code paths that use these parameters, finding the vulnerabilities. However, this should not overshadow the broader perspective in terms of trade-offs. First, when looking at vulnerable applications, half of the applications found vulnerable via static analysis were also detected through indicators. Second, indicators excel in testing a wider range of applications potentially at risk, which static analysis alone might overlook. Consequently, indicators can play a crucial role in **complementing** static analysis, helping to cast a wider net and pinpointing applications that warrant a more in-depth examination.

V-C2 Analysis Time. The main benefit of vulnerability mining over static analysis is highlighted by the significant differences in runtime costs. Specifically, running the JAW static analysis pipeline to construct a CPG and execute analysis queries took an average of 34m 52s for one webpage, and about 1,024 days for the whole P_2' (we used 100 parallel executions to do this in 10 days). Accordingly, running static analysis for the entire P_2 would require an estimated 12,789 days with sequential execution (or $\sim$127 days by 100 parallel instances). In stark contrast, mining all indicator patterns on entire P_2' and P_2 were accomplished in about 29m and 58m, respectively [2].

V-C3 Verification Time. The verifier performs between one and 26 tests per URL and needs 10 seconds per test. On P_2', verifying static analysis results took about 1.5h, whereas indicators needed 217h (or $\sim$2h with 100 parallel executions).

V-C4 Storage Requirements. In terms of storage needs, static analysis demonstrated a considerably higher demand as well. The average size of a CPG and its corresponding query results was 29.5 MB, cumulatively amounting to 1.1 TB for the P_2' dataset. Extrapolating these figures, processing the P_2 dataset is estimated to require about 14.8 TB of disk space. On the other hand, the vulnerability mining method required only 25 GB of space for the entire P_2.

These comparisons highlight the substantial advantages of vulnerability mining in terms of both speed and resource utilization, making it a suitable **trade-off** for larger-scale security assessments, which can provide a lowerbound on the number of affected sites.

VI. Exploit Analysis and Escalation

Starting from the vulnerabilities we discovered in Sections IV and V, we now examine their susceptibility to more critical exploitation scenarios including XSS, information leakage, and request forgery attacks, as described in our threat model of §II-B. Particularly, we discovered a total of 20,898 open redirections across 623 sites in §IV through static-dynamic program analysis, and 375 vulnerabilities within 326 sites in §V through vulnerability mining, summing to 21,273 vulnerabilities across 872 unique websites.

A. Methodology

VI-A1 Cross-Site Scripting. To assess the potential for DOM-based XSS exploitations, we employed an automatic approach, where we tested the susceptibility of each vulnerable endpoint against a subset of the attack techniques outlined in Table X, specifically those capable of leading to DOM-based XSS. For each vulnerability, we loaded the webpage in Playwright [43], inserted the attack payload at the injection point, and verified whether the payload executed as intended. We note that we used a benign attack payload based on the ``debugger;" JavaScript instruction, which serves as a breakpoint pausing the execution of client-side code.

[2]Runtimes are based on the following configuration: Ubuntu 18.04, AMD EPYC 7H12 processor with 256 CPU cores and 2 TB RAM.

Threat	SAST Vuln.	SAST Sites	Mining Vuln.	Mining Sites	Total Vuln.	Total Sites
DOM-based XSS	1,845	212	84	78	1,929	290
Client-side CSRF	36	33	6	6	42	39
Information Leak	2	2	1	1	3	3
Total	1,883	**247**	91	**85**	1,974	**332**

TABLE V: Summary of exploitations created for open redirect vulnerabilities. SAST and mining refer to steps 1 and 2 of Figure 2, respectively.

VI-A2 Request Forgery and Information Leakage. To examine potential escalations to request forgery and information leakage attacks, we employed a manual approach. Due to the large number of confirmed vulnerabilities—21K open redirects across 872 websites—it was infeasible to manually create xploits for each one. Instead, we focused on demonstrating the potential for escalation by examining a random subset, where we aimed to maximize the coverage across various sites. Therefore, we randomly selected up to two vulnerabilities from each of the 872 affected websites, giving us a total of 1,744 vulnerabilities to analyze.

For each attack scenario, we conducted specific tests. 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 redirect request for the presence of sensitive data like authorization keys, and OAuth tokens. Due to ethical considerations, we excluded testing requests and functionalities where we could not control the impact (e.g., publicly accessible content), and use our own test accounts exclusively.

We note that identifying request forgery and information leakage exploits automatically poses a non-trivial challenge, demanding a deep understanding of each specific application to pinpoint target endpoints for request forgery considering the request semantics, or the presense of sensitive information. Moreover, it involves assessing whether the client-side requests induce server-side state changes. Finally, in an automated setting, guaranteeing ethical compliance and preventing unintended server-side interactions or state changes is challenging. For these reasons, we opted for a systematic manual approach where we can strictly control our tests.

B. Results

We now provide an overview of the exploitation results for the vulnerabilities, following the methodology in §VI-A.

VI-B1 Cross-Site Scripting. In total, we automatically tested DOM-based XSS escalations for 21,273 open redirects across 872 as shown in Sections IV and V. Our analysis revealed that about 9% of the vulnerabilities across 33.2% of the affected

Listing 2: Open redirect vulnerability in *adobe.com* escalated to DOM XSS.

```
1  class i {
2      constructor(n, /* [...] */){
3          this._injector = n;
4          /* [...] */}
5      navigate(n){
6          const w = this._injector.get("w"),
7          x = n.queryParamMap.get("externalUrl"),
8          k = n.queryParamMap.get("windowTarget"),
             // _self
9          d = n.queryParamMap.get("rel");
10         if(x.indexOf('adobe.com') != -1)
11             w.open(x, k, d);}}
12 n = {};
13 s = location.search;
14 n.queryParamMap = new URLSearchParams(s);
15 n.w = window;
16 x = new i(n);
17 x.navigate(n);
```

sites could be escalated to DOM-based XSS attacks, which is alarming. When comparing static analysis-based and indicator-based approaches, static analysis identified approximately two orders of magnitude more vulnerabilities across nearly double the number of sites. However, only around 8% of these vulnerabilities could be exploited for XSS. In contrast, more than 22% of the vulnerabilities identified through the indicator-based approach could be exploited for XSS, indicating a higher prevalence of XSS escalations among indicator-based findings.

VI-B2 Request Forgery and Information Leakage. In total, we discovered 42 client-side CSRF and three cross-site information leakage vulnerabilities, suggesting that over 2.4% and only about 0.2% of the open redirects can be escalated to request forgery and information leaks. Despite their relatively lower incidence compared to XSS, these exploitations could still led to critical consequences like account takeover and unauthorized changes to account settings, compromising the integrity of the applications' databases. Table V provides a summary of our findings.

C. Case Studies

We present a few manually vetted case studies of the confirmed attacks (disclosed and patched), with additional case studies in Appendix B,

Adobe. Listing 2 shows a DOM XSS exploitation of a client-side open redirect vulnerability in *adobe.com*. The vulnerability originates in line 11, where the code employs the `window.open()` API to redirect the current window to a destination controlled by the attacker, which is read from the top-level URL query parameters, specifically the `externalUrl` key. The code attempts input validation in line 10 by checking if the destination string contains the string 'adobe.com' using the `indexOf()` function. However, this check is insufficient: (i) an attacker can achieve open redirection to a domain like `adobe.com.attack.com`, bypassing the `indexOf()` check, and (ii), there is no check against the `javascript:` scheme, enabling attackers to escalate it to a DOM XSS.

Listing 3: CSRF escalation of an open redirect vulnerability in *webnovel.com*.

```
1  /* extract a query parameter value from URL */
2  function u(e) {
3    r = new RegExp("(^|&)" + e + "=([^&]*)(&|$)",
       "i"),
4    t = new RegExp("[A-Za-z]"),
5    n = window.location.search.substr(1).match(r);
6    if (null != n) {
7      var o = n[2];
8      return t.test(o) ? n[2] : parseInt(n[2]);}
9    return null; }
10 /* check destination URL */
11 function isValidUrl(e) {
12   a =  !!/^\/[^/]*/.test(e); //
       protocol-relative URIs
13   b =  !!e.match(/^https?:\/\/[^.]*?\.webnovel\.
       com($|\/.*|\?|#)/);
14   return a || b;}
15 /* redirection */
16 var c = {
17   code: u("code"),
18   ticket: u("ticket"),
19   guid: u("userid"),
20   forceRedirect: u("forceRedirect"),
21   redir: decodeURIComponent(u("redirectUrl") ||
       ""))}
22 r = c.redir;
23 r && isValidUrl(r)? location.assign(r):
24     location.href = "/";
```

WebNovel. Listing 3 illustrates a simplified open redirect vulnerability in *webnovel.com* that we escalated to client-side CSRF. The open redirection takes place in line 23 using the `location.assign()` API whose parameter `r` is retrieved from the `redirectUrl` query parameter in line 21 using the function `u` defined in line 2. The code validates the variable `r` in line 23 using the `isValidUrl` function, checking if the destination satisfies one of the two properties: (i) it starts with `//`, or (ii) it belongs to the *webnovel.com* domain. Condition (i) allows open redirection abusing protocol-relative URIs, e.g., `//attack.com`, whereas, XSS exploitation is not possible as `javascript` URIs are not allowed. However, further investigation revealed that WebNovel employs state-changing GET requests to save modifications to user account settings. This discovery enabled us to forge the redirection request, establishing a client-side CSRF attack vector that empowers attackers to manipulate user account settings. We note that the advantage of a client-side open redirect is that it triggers a top-level, same-site request, resulting in client-side CSRF [18, 26], compared to cross-site requests in traditional CSRF attacks. State-changing GET requests triggered via cross-site resources are prevented by SameSite cookies [39]. However, GET requests triggered via client-side open redirects can bypass SameSite cookie protections.

VK. We identified an open redirect vulnerability in *vk.com*, which can be escalated to DOM-based XSS. The vulnerable URL includes a query key `to`, specifying the destination for the final redirect (e.g., `attack.com`). Upon receiving such a request, the application sets a cookie, `remixsec_redir=attack.com`, through the `Set-Cookie` HTTP response header. Subsequently, the

Tool	Ref.	Method	Vuln.	Conf.	FP	FN
Joern: v1.1.1277	[80, 82]	Static	11	2	9	1
JAW: v1	[18]	Static	5	3	2	0
JAW: v2 (TheThing)	[19]	Hybrid	5	3	2	0
JAW: v3 (Sheriff)	[16]	Hybrid	4	3	1	0
BlackWidow: v1.3	[81]	Dynamic	1	1	0	2
Foxhound: v98.0.2	[44, 45]	Dynamic	72	2	70	1
STORK		Hybrid	3	3	0	0

TABLE VI: Comparison of indicator-based vulnerability scanning with XSS detectors. **Legend:** Vuln= potential vulnerabilities; Conf.= manually confirmed; FP= false positive; FN= false negative.

Mitigation	Domains	Pct.
#1: Redirect Notice	2,178	54.4%
#2: Input Validation	1,051	26.2%
#3: Content Security Policy	416	10.4%
#4: Security Tokens	112	2.8%
#5: Captcha / reCaptcha	43	1.0%
#6: Link Shimming	14	0.3%
No Redirect	186	4.6%

TABLE VII: Mitigation techniques employed by websites.

client-side code reads the value of the `remixsec_redir` cookie and additionally checks if the URL contains another query parameter, `away`, with a non-empty token value. If this condition is met, it redirects the current page to the value of `remixsec_redir` (i.e., `attack.com` or a `javascript` URI). This way, an attacker can also implant a persistent DOM XSS attack vector, and exploit it only later on to attack a victim (i.e., when the `away` parameter is present).

D. Comparison with XSS Detectors

We compared DOM XSS detection between indicator-based vulnerability scanning described in §VI-A and XSS detection methods on the P_2' dataset in §V-C (42K webpages of 50 random applications). We considered the following state-of-the-art detectors as baselines: dynamic taint-tracking [24, 36, 45] using Foxhound [44, 45], JAW engine versions one to three [16, 18, 19], Joern [80], and BlackWidow [81].

Table VI summarizes the results. Overall, the indicator-based approach and static analysis based on JAW found three DOM XSS vulnerabilities. In contrast, other approaches discovered less true positive vulnerabilities. Specifically, Foxhound identified 72 potentially sensitive data flows, but only two were confirmed as XSS after manual analysis. The high FP rate stems from the fact that Foxhound only detects the presense of data flows that may lead to XSS but does not verify whether these flows are actually attacker-controlled. Foxhound missed one XSS data flow, as it could not trigger the vulnerable execution path (branches). BlackWidow identified only one true positive vulnerability because its test payloads failed to detect other vulnerable injection points. Finally, Joern did not find one XSS vulnerability due to a missing call graph edge for the `Function.call()` instruction.

VII. Defenses in the Wild

We identify and study the various mitigation techniques deployed in the wild to address open redirect vulnerabilities. We intend to have a look at the subset of candidate redirection URLs found via indicator pattern mining that were not vulnerabilities (e.g., closed redirections) and investigate potential mitigations employed by websites.

Starting from the results in Table XI, there are 7,719 websites where we identified a candidate URL matching our indicators, but we did not observe an open redirect during automatic run-time experiments (refer to §V-B). To identify the mitigation strategy, we randomly selected 4K of the 7.7K sites, and investigated one random candidate URL per site semi-automatically.

The examination of the 4K sites revealed six different mitigation techniques, outlined in Table VII. We found that 57% of the cases (i.e., 2,292 sites) used client-side redirects, whereas 38% of sites employed server-side redirection, and in the remaining 5%, we did not observe any redirection. In the following, we discuss each mitigation technique.

Redirect Notice. We observed that more than half of the non-vulnerable sites display a redirection warning to the user. In the majority of these cases (i.e., 84.5%), intermediate human interaction is necessary before the redirection occurs, such as a button click or entering an input. However, in the remaining 15.5% of cases, the redirection happens automatically after a certain period (e.g., 120 seconds), and the user can only expedite the redirection by, for example, clicking on a button. While the risk is negligible, it's worth noting that these cases could lead to open redirection after a certain amount of time automatically if the user leaves the webpage open. To determine if a webpage had a redirect notice, we loaded the page via Playwright using an automated script, and then vet manually whether we can see a notice, after which our script opens up the next site.

Captcha. In addition to redirect notices, we observed another form of intermediate interaction utilized by websites before redirecting users to specified targets: captchas. We found that about 1% of the sites display a captcha page before redirection, including popular sites like Amazon. To identify these cases, we followed a similar methodology as that of redirect notices.

Input Validation. These checks involve examining user-provided data to ensure it adheres to expected formats or constraints before using it as a part of the redirect destination [45, 83]. To detect the presence of these checks, we tested the redirection automatically by setting a URL belonging to the domain of the site under test. If the redirection happens automatically, we confirm the presence of a input validation routine, as it did not work for an external domain in our previous analysis. Also, we manually looked at the client-side code for about 10% of these pages to identify the variety of checks happening in client-side. Overall, our analysis revealed that more than a quarter of the non-vulnerable websites (i.e., 1,051) implemented input validation, including various validation checks in client-side JavaScript program inputs, such as hard-coded equality conditionals, whitelists, length validations, checks for data types and formats, use of regular expressions for pattern matching, input sanitization routines to filter out potentially malicious JavaScript content, URL substring searches, and other string manipulation and comparison operations.

Content Security Policy (CSP). CSP [27] can mitigate the impact of client-side open redirects when attackers can control the value of JavaScript instructions that trigger page navigation, such as `location.assign()` API [84]. For example, CSP can be configured to restrict the domains to which a page can be redirected, particularly using the `navigate-to` directive [85], thereby mitigating the risk of redirections to external domains. We collected the CSP policies automatically and confirmed that over 10% of the non-vulnerable sites adopt a CSP policy blocking the redirection.

Link shimming. Link shimming [86] refers to a technique where an application transforms its URLs in a way that allows it to intercept and analyze the traffic before redirecting the user to the intended destination. When a request is intercepted, the service cross-checks the URL against internal lists of malicious domains and external partners' lists. Then, the service redirects the user to an intermediate page to confirm the redirection, similarly to a redirect notice, and if the request seems suspicious, warns the user about it. We identified 14 websites in our dataset (i.e., 0.3%) with this behaviour, the majority of which are social media platforms. For example, link shim traffic for Facebook is transferred to `l.facebook.com`.

Security Tokens. We discovered that a small portion of the sites (2.8%) employed tokens to allow or block redirections to external domains. These are cryptographic tokens, nonces that expire after a single use, or time-synchronized tokens that remain valid for a specific period, allowing multiple uses within that timeframe before expiration. While the risk is marginal, the latter cases could still be abused for open redirections within their small validity window, affecting popular sites like AliExpress and Samsung.

VIII. Related Work

Open Redirect Vulnerabilities. Unvalidated redirects have been the focus of several research efforts in the past. Shue et. al. [2] presented the first set of manually-curated heuristics to identify potential open redirect vulnerabilities and used dynamic analysis to test them. The authors demonstrated that open redirects were ubiquitous in the wild back in 2008. Almost seven years later, Wang et. al. [3] quantified the prevalence of unvalidated redirects leveraging a custom black-box scanner and showed that many websites are still affected by this security flaw. Since then, open redirect vulnerabilities have been attracting the attention of the security community, with researchers exploring the attack surface [72, 75, 87], testing strategies [56, 59, 73], and mitigation techniques [28–31, 74]. More recently, multiple works [17, 88] studied security flaws affecting OAuth 2.0 implementations concerning the validation of the redirect URI parameter. In parallel with prior research efforts, our study reveals that unvalidated redirects persist as a prevalent security concern in the wild, even after nearly a decade. Our work goes beyond mere prevalence measurement by extracting vulnerable patterns and showcasing how attackers can escalate these vulnerabilities. In addition, we demonstrate how adversaries can actively search for these vulnerabilities in public data repositories.

Indicator-based Vulnerability Discovery. Scanning programs for indicators to identify potential vulnerabilities has been considered by several researchers in the past (e.g., [89–92]). Broadly, we can divide these techniques into two classes: metric-based and pattern-based techniques. Metrics-based approaches use machine learning models to predict vulnerable code locations in the source code, using features like static and execution code complexity [93, 94], token frequency [95, 96], dependency relationships [97], and developer activity metrics [98–100]. These approaches are typically heavy and applying them to the context of web applications requires training datasets. Instead, we focused on lightweight approaches next to costly static analysis to enable larger-scale analyses. Conversely, pattern-based techniques rely on syntax and semantics of vulnerable programs to extract a pattern, which is used to identify potentially vulnerable code, typically through static analysis [19, 46, 101, 102]. However, existing pattern-based solutions for open redirects [2, 3] propose hand-crafted lists of indicators. In contrast, we automatically extract patterns using a novel, static-dynamic methodology.

Program Analysis for Security Testing. The field of program analysis for security testing has witnessed significant attention in the last decade. Researchers proposed various techniques to examine the security posture of software applications, including static analysis [46, 103, 104], dynamic analysis [24, 36, 37, 45, 55, 63, 105], and hybrid approaches [18, 19, 106, 107]. For example, Lekies et al. [24] modified the JavaScript engine in Chromium to enhance it with taint-tracking capabilities, focusing on the detection of DOM-based XSS. Melicher et. al. [25] adopted a similar methodology, but employed a new strategy to verify if a data flow could indicate a DOM XSS vulnerability. Similarly, Steffens et al. investigated the prevalence of persistent [37] and postMessage-based XSS [55] through dynamic taint tracking and forceful execution, respectively. Klein et al. [45] adopted a combined approach involving dynamic taint tracking and symbolic string analysis, focusing on the robustness of custom sanitization functions deployed on the Web. Khodayari and Pellegrino proposed a hybrid system, JAW [18], to study client-side CSRF vulnerabilities in JavaScript programs. Saxena et al. introduced Kudzu [106], a tool that combines taint tracking and symbolic execution to detect source-sink data flows in the client-side of web applications. Other works studied code-less injection attacks using both static and dynamic approaches, such as script gadgets [36], DOM Clobbering [19], and mutation-based XSS [108]. Our work uses and extends these techniques by applying them to the problem of open redirections on the Web.

IX. Conclusion and Discussion

In this section, we summarize our findings and discuss their wider implications.

A. *Re-evaluating the Risk and Lessons Learned*

This section re-evaluates the risk posed by open redirects by contextualizing our findings alongside previous research. However, we note that a direct comparison with prior work is challenging because of potentially different methodologies, tools, and snapshots of the Web.

Prevalence. Our study reveals that open redirect vulnerabilities are widespread, impacting approximately 8.7% of the top 10K websites, totaling over 21.2K instances. In comparison, recent research [16] discovered that client-side CSRF vulnerabilities affect about 9% of the websites, with over 72.3K instances, whereas code-less injection attacks, particularly, DOM Clobbering [19] and script gadgets [36] are present on 9.8% (9.4K instances) and 19.8% (285.8K instances) of top 5K sites, respectively. Accordingly, client-side open redirect vulnerabilities have a high incidence rate similarly to other client-side web vulnerability classes.

Impact and the Role of Modern Redirection APIs. Our work shows that modern JS redirection APIs can lead to significant vulnerabilities, challenging the long-standing perception of open redirects as low-impact. Specifically, we showed that $\sim$38% of sites that have an open redirect vulnerability (i.e., 3.3% of the top 10K sites) can be leveraged for more critical attacks. For example, JS redirection APIs trigger top-level requests, thereby bypassing SameSite cookie protections for CSRF, and support the `javascript:` request scheme, thereby posing the risk of XSS. Also, our work shows that attackers can abuse fast indicator searches to find such high-impact open redirects with comparatively less effort.

Severity. We found that almost one out of ten open redirects can be escalated to DOM-based XSS, which corresponds to about 2.9% of the top 10K websites. In comparison, recent research by Melicher et. al. [25] showed that $\sim$3.6% of the top 10K domains are vulnerable to DOM-based XSS through dynamic taint tracking. We showed that the impact of open redirect extends even further, as we exploited 2.6% of the open redirections for request forgery and information leakage attacks, demonstrating an alarming landscape.

B. *Effective Defenses*

Our study reveals that a significant fraction of websites with closed redirections ($\sim$54%) incorporate an intermediate human interaction step, such as a redirect notice, to warn users of the redirection request. While redirect notices are beneficial, their efficacy is threatened by the risky and rather common practice of time-budgeted notices, whose expiration results in an automatic redirect (occurring in $\sim$15% of cases). Furthermore, they are not sufficient to prevent redirections to JavaScript-based URIs, leaving the potential for escalations to DOM XSS attacks in the case of client-side redirects.

Our findings indicate that $\sim$25% of the websites with closed redirects validate the destination of redirections, addressing, among others, the risk of DOM XSS. Furthermore, an additional 10% implement a CSP policy that can act as a defense-in-depth and mitigate the impact of a potential DOM XSS exploitation, which is promising. However, we also observed that $\sim$13.2% of the sites that found to be vulnerable to open redirect via static analysis and that can also be escalated to XSS (i.e., 28 out of 212), have indeed adopted a CSP policy in another webpage but not the vulnerable one, which could have mitigated XSS exploitation, further highlighting the importance of consistent adoption of security policies across webpages [39, 109, 110]).

Finally, we observed that the efficacy of defenses is largely influenced by the use cases of redirections in web applications. Validating destinations for redirection services such as link shorteners [111], advertisement services [112] or social networks, which serve diverse purposes, presents greater challenges (see, e.g., [86]) compared to application-specific functionalities with a restricted set of possible redirection endpoints, such as post-login redirects or OAuth redirection URIs [17]. For instance, while such services often need to omit HTTP `referer` headers to mitigate the risk of privacy issues or cross-domain information leakage [113, 114], application-specific redirections may require the use and forwarding of the `referer` to implement defense-in-depth, e.g., for CSRF [115].

C. *Ethical Considerations*

Our experiments on live websites are limited to user accounts that we created exclusively for this purpose, such as manual tests for state-changing operations. During our testing process, we followed the guidelines provided by the website's vulnerability disclosure programs on Bugcrowd [116] and HackerOne [117] to maintain testing transparency and uphold responsible research practices. To mitigate any potential impact on resource servers during dynamic analysis, we minimized the testing load by implementing a round-robin strategy, where we tested a single URL from each unique domain before moving on to the next domain. Additionally, we rigorously imposed limits on our testing, conducting a maximum of 250 requests per day and domain. We responsibly disclosed all the vulnerabilities we detected to the affected parties following best practices [118]. We provide details about our notification campaign in Appendix A.

D. *Open Science*

We publicly release our catalog of vulnerability indicators and the STORK framework[3].

E. *Weighing the Testing Trade-Offs*

In this paper, we showed the trade-offs between static analysis and indicator-based scanning to detect open redirects. We found that while static analysis can identify more vulnerabilities and is more precise, it also requires significantly more

[3]`https://github.com/SoheilKhodayari/STORK`

time and storage. In comparison, indicator-based approaches are roughly 100 times faster and need 13 times less storage, demonstrating their benefit in scaling up. Indicators can detect vulnerabilities missed by static analysis, and they can play a crucial complementary role. Notably, half of the applications identified as vulnerable through static analysis were also detected by indicators, helping to pinpoint applications that need more in-depth testing. Given that the cost of indicator-based mining is significantly lower than fully-fledged static and dynamic analysis, and considering their concerning exploitation potential as shown in §VI, adversaries can actively search for these vulnerabilities in public data repositories.

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] (2024) Redirection 3xx HTTP response code. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7231#section-6.4

[2] C. A. Shue, A. J. Kalafut, and M. Gupta, "Exploitable redirects on the web: Identification, prevalence, and defense," in *USENIX WOOT*, 2008.

[3] J. Wang and H. Wu, "Urfds: Systematic discovery of unvalidated redirects and forwards in web applications," in *IEEE CNS*, 2015.

[4] MITRE Open Redirect CVEs. [Online]. Available: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=open+redirect

[5] D. Canali, M. Cova, G. Vigna, and C. Kruegel, "Prophiler: a fast filter for the large-scale detection of malicious web pages," in *Proceedings of the 20th international conference on World wide web*, 2011.

[6] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad, "Towards measuring and mitigating social engineering software download attacks," in *USENIX Security Symposium*, 2016.

[7] GreatHorn. (2021) Google and Open Redirects: Preventing Your Users from Becoming a Victim of Attacks. [Online]. Available: https://www.greathorn.com/blog/google-and-open-redirects-preventing-your-users-from-becoming-a-victim-of-attacks/

[8] A. Oest, P. Zhang, B. Wardman, E. Nunes, J. Burgis, A. Zand, K. Thomas, A. Doupé, and G.-J. Ahn, "Sunrise to sunset: Analyzing the end-to-end life cycle and effectiveness of phishing attacks at scale," in *USENIX Security Symposium*, 2020.

[9] C. Whittaker, B. Ryner, and M. Nazif, "Large-scale automatic classification of phishing pages," 2010.

[10] (2024) Google and Alphabet Vulnerability Reward Program (VRP) Rules. [Online]. Available: https://bughunters.google.com/about/rules/6625378258649088/google-and-alphabet-vulnerability-reward-program-vrp-rules

[11] (2024) Microsoft M365 Bounty Program. [Online]. Available: https://www.microsoft.com/en-us/msrc/bounty-online-services

[12] (2022) Redirect url vulnerable to XSS and open redirect. [Online]. Available: https://www.bugbountyhunter.com/hackevents/report?id=1498

[13] (2021) Chaining open redirect with XSS to account takeover. [Online]. Available: https://rdnzx.medium.com/chaining-open-redirect-with-xss-to-account-takeover-36acf218a6d5

[14] (2023) OWASP DOM Clobbering prevention cheat sheet. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/DOM_Clobbering_Prevention_Cheat_Sheet.html

[15] Detectify. (2019) The real impact of an Open Redirect vulnerability. [Online]. Available: https://blog.detectify.com/2019/05/16/the-real-impact-of-an-open-redirect/

[16] S. Khodayari, T. Barber, and G. Pellegrino, "The great request robbery: An empirical study of client-side request hijacking vulnerabilities on the web," in *Proceedings of 45th IEEE Symposium on Security and Privacy*, 2024.

[17] T. Innocenti, M. Golinelli, K. Onarlioglu, A. Mirheidari, B. Crispo, and E. Kirda, "Oauth 2.0 redirect uri validation falls short, literally," in *Proceedings of the 39th Annual Computer Security Applications Conference*, 2023.

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

[19] ——, "It's (DOM) Clobbering Time: Attack Techniques, Prevalence, and Defenses," in *IEEE S&P Symposium*, 2023.

[20] Google Custom Search JSON API. [Online]. Available: https://developers.google.com/custom-search/v1/overview

[21] F. Toffalini, M. Abbà, D. Carra, and D. Balzarotti, "Google dorks: Analysis, creation, and new defenses," in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), San Sebastián*, 2016.

[22] Google Dorking Cheatsheet. [Online]. Available: https://github.com/chr3st5an/Google-Dorking

[23] Wayback Machine APIs. [Online]. Available: https://archive.org/help/wayback_api.php

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

[25] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, "Riding out domsday: Towards detecting and preventing dom cross-site scripting," in *Network and Distributed System Security Symposium*, 2018.

[26] (2018) Client-side CSRF. [Online]. Available: https://www.facebook.com/notes/facebook-bug-bounty/client-side-csrf/2056804174333798/

[27] M. West, "Content Security Policy Level 3," *W3C Working Draft*, 2024. [Online]. Available: https://w3c.github.io/webappsec-csp/

[28] OWASP Unvalidated Redirects and Forwards Cheat Sheet. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html

[29] Portswigger, "Open redirections," 2022. [Online]. Available: https://portswigger.net/kb/issues/00500100_open-redirection-reflected

[30] J. Galloway. (2022) Microsoft: Preventing Open Redirection Attacks (C#). [Online]. Available: https://docs.microsoft.com/en-us/aspnet/mvc/overview/security/preventing-open-redirection-attacks

[31] Google. (2009) Open redirect URLs: Is Your Site Being Abused? [Online]. Available: https://developers.google.com/search/blog/2009/01/open-redirect-urls-is-your-site-being

[32] (2022) The HTTP Location Response Header. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Location

[33] R. Auger. (2011) WASC Article on URL Redirector Abuse. [Online]. Available: http://projects.webappsec.org/w/page/13246981/URL%20Redirector%20Abuse

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

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

[36] 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.

[37] 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.

[38] Stealing OAuth Tokens With Open Redirects. [Online]. Available: https://sec.okta.com/articles/2021/02/stealing-oauth-tokens-open-redirects

[39] 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.

[40] 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.

[41] E. Wang, J. Chen, W. Xie, C. Wang, Y. Gao, Z. Wang, H. Duan, Y. Liu, and B. Wang, "Where urls become weapons: Automated discovery of ssrf vulnerabilities in web applications," in *IEEE Symposium on Security and Privacy*, 2024.

[42] G. Pellegrino, O. Catakoglu, D. Balzarotti, and C. Rossow, "Uses and abuses of server-side requests," in *19th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2016.

[43] Playwright browser automation framework. [Online]. Available: https://playwright.dev/

[44] Project Foxhound. [Online]. Available: https://github.com/SAP/project-foxhound

[45] 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.

[46] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in *IEEE S&P Symposium*, 2014.
[47] Neo4j Graph Database. [Online]. Available: https://neo4j.com/
[48] Cypher Query Language. [Online]. Available: https://neo4j.com/developer/cypher/
[49] Arrow function expressions. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions.
[50] setInterval global function. https://developer.mozilla.org/en-US/docs/Web/API/setInterval.
[51] globalThis Object. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/globalThis.
[52] Promise.prototype.then(). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then.
[53] M. Madsen, O. Lhoták, and F. Tip, "A model for reasoning about javascript promises," in *ACM OOPSLA*, 2017.
[54] Web API Specifications. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API
[55] M. Steffens and B. Stock, "PMForce: Systematically Analyzing postMessage Handlers at Scale," in *CCS*, 2020.
[56] Using Burp to Test for Open Redirections. [Online]. Available: https://portswigger.net/support/using-burp-to-test-for-open-redirections
[57] BurpSuite. Last accessed June 2024. [Online]. Available: https://portswigger.net/burp
[58] (2010) Owasp zed attack proxy. https://www.zaproxy.org/.
[59] ZAP: Open Redirect. [Online]. Available: https://www.zaproxy.org/docs/alerts/10028/
[60] D. W. Binkley and K. B. Gallagher, "Program slicing," *Advances in computers*, 1996.
[61] window.open() API. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/open
[62] window.name API. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/name
[63] S. Son and V. Shmatikov, "The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2013.
[64] S. A. Mirheidari, M. Golinelli, K. Onarlioglu, E. Kirda, and B. Crispo, "Web cache deception escalates," in *USENIX Security Symposium*, 2022.
[65] O. Tsai, "A New Era of SSRF - Exploiting URL Parser in Trending Programming Languages," *Blackhat USA*, 2017. [Online]. Available: https://www.blackhat.com/docs/us-17/thursday/us-17-Tsai-A-New-Era-Of-SSRF-Exploiting-URL-Parser-In-Trending-Programming-Languages.pdf
[66] N. Gruegoire, "Server-Side Browsing Considered Harmful," *OWASP AppSec EU, Amsterdam*, 2015. [Online]. Available: https://www.agarri.fr/docs/AppSecEU15-Server_side_browsing_considered_harmful.pdf
[67] N. Moshe, S. Brizinov, R. Onitza-Klugman, and K. Efimov, "Exploiting url parsers: the good, the bad and the inconsistent," 2021.
[68] HackerOne Open Redirect Vulnerability Reports. [Online]. Available: https://hackerone.com/hacktivity/overview?queryString=open+redirect+AND+disclosed%3Atrue
[69] CWE-601: URL Redirection to Untrusted Site. [Online]. Available: https://cwe.mitre.org/data/definitions/601.html
[70] StackExchange Security Community. [Online]. Available: https://security.stackexchange.com/
[71] Dev Security Community. [Online]. Available: https://dev.to/t/security
[72] C. Polop. (2022) HackTricks Cheatsheet Series - Open Redirect. [Online]. Available: https://book.hacktricks.xyz/pentesting-web/open-redirect
[73] OWASP Testing Guide for Client-side URL Redirects. [Online]. Available: https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/11-Client-side_Testing/04-Testing_for_Client-side_URL_Redirect
[74] P. Schulz. (2022) intigriti Open Redirect Article. [Online]. Available: https://blog.intigriti.com/hackademy/open-redirect/
[75] V. Security, "URL Redirection: Attack and Defense," 2022. [Online]. Available: https://www.virtuesecurity.com/kb/url-redirection-attack-and-defense/
[76] Open Redirect Filter Bypass Methods. [Online]. Available: https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/Open%20Redirect
[77] Payloads from Bug Bounty Reports for Open Redirect. [Online]. Available: https://github.com/cujanovic/Open-Redirect-Payloads
[78] 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.
[79] Google Programmable Search Engine Pricing. [Online]. Available: https://developers.google.com/custom-search/docs/overview
[80] Joern jssrc2cpg library. [Online]. Available: https://github.com/joernio/joern/tree/master/joern-cli/frontendjssrc2cpg
[81] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black widow: Blackbox data-driven web scanning," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.
[82] Joern engine. [Online]. Available: https://github.com/joernio/joern
[83] M. Alkhalaf, T. Bultan, and J. L. Gallegos, "Verifying client-side input validation functions using string analysis," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012.
[84] Location: assign() method. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Location/assign
[85] Content Security Policy: navigate-to directive. [Online]. Available: https://csplite.com/csp123/
[86] F. Li, "Shim shimmeny: evaluating the security and privacy contributions of link shimming in the modern web," in *USENIX Security Symposium*, 2020.
[87] Pentester Land Open Redirect Cheetsheet. [Online]. Available: https://pentester.land/cheatsheets/2018/11/02/open-redirect-cheatsheet.html
[88] R. Yang, G. Li, W. C. Lau, K. Zhang, and P. Hu, "Model-based security testing: An empirical study on oauth 2.0 implementations," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016.
[89] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, "Leopard: Identifying vulnerable code for vulnerability assessment through program metrics," in *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019.
[90] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira, "Vulnerable code detection using software metrics and machine learning," *IEEE Access*, 2020.
[91] F. Yamaguchi, K. Rieck *et al.*, "Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning," in *5th USENIX Workshop on Offensive Technologies (WOOT 11)*, 2011.
[92] K. Z. Sultana, V. Anu, and T.-Y. Chong, "Using software metrics for predicting vulnerable classes and methods in java projects: A machine learning approach," *Journal of Software: Evolution and Process*, 2021.
[93] S. Moshtari, A. Sami, and M. Azimi, "Using complexity metrics to improve software security," *Computer Fraud & Security*, vol. 2013, 2013.
[94] Y. Shin and L. Williams, "An initial study on the use of execution complexity metrics as indicators of software vulnerabilities," in *Proceedings of the 7th International workshop on software engineering for secure systems*, 2011.
[95] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, 2014.
[96] Y. Zhang, D. Lo, X. Xia, B. Xu, J. Sun, and S. Li, "Combining software metrics and text features for vulnerable file prediction," in *2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2015.
[97] V. H. Nguyen and L. M. S. Tran, "Predicting vulnerable software components with dependency graphs," in *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, 2010.
[98] M. Gegick, L. Williams, J. Osborne, and M. Vouk, "Prioritizing software security fortification throughcode-level metrics," in *Proceedings of the 4th ACM workshop on Quality of protection*, 2008.
[99] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE transactions on software engineering*, 2010.
[100] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Software Engineering*, 2013.
[101] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proceedings of the 28th annual computer security applications conference*, 2012.
[102] J. Vanegue and S. K. Lahiri, "Towards practical reactive security audit using extended static checkers," in *2013 IEEE Symposium on Security and Privacy*, 2013.
[103] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi,

"Efficient and Flexible Discovery of PHP Application Vulnerabilities," in *Proceedings of the 2nd IEEE European Symposium on Security and Privacy*, 2017.
[104] F. Al Kassar, G. Clerici, L. Compagna, D. Balzarotti, and F. Yamaguchi, "Testability Tarpits: the Impact of Code Patterns on the Security Testing of Web Applications." in *NDSS Symposium*, 2022.
[105] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow, "Deemon: Detecting CSRF with dynamic analysis and property graphs," in *ACM CCS*, 2017.
[106] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *IEEE Symposium on Security and Privacy*, 2010.
[107] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan, "NAVEX: Precise and scalable exploit generation for dynamic web applications," in *USENIX Security Symposium*, 2018.
[108] 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.
[109] S. Calzavara, T. Urban, D. Tatang, M. Steffens, and B. Stock, "Reining in the Web's Inconsistencies with Site Policy," in *Network and Distributed Systems Security Symposium*, 2021.
[110] A. Mendoza, P. Chinprutthiwong, and G. Gu, "Uncovering HTTP Header Inconsistencies and the Impact on Desktop/Mobile Websites," in *World Wide Web Conference*, 2018.
[111] F. Klien and M. Strohmaier, "Short links under attack: geographical analysis of spam in a url shortener network," in *Proceedings of the 23rd ACM conference on Hypertext and social media*, 2012.
[112] P. Papadopoulos, N. Kourtellis, and E. P. Markatos, "The cost of digital advertisement: Comparing user and advertiser views," in *World Wide Web Conference*, 2018.
[113] Cross-domain referer leakage. [Online]. Available: https://portswigger.net/kb/issues/00500400_cross-domain-referer-leakage
[114] B. Krishnamurthy, K. Naryshkin, and C. Wills, "Privacy leakage vs. protection measures: the growing disconnect," in *Proceedings of the W2SP Conference*, no. 2011, 2011.
[115] X. Likaj, S. Khodayari, and G. Pellegrino, "Where we stand (or fall): An analysis of csrf defenses in web frameworks," in *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, 2021.
[116] Bugcrowd. [Online]. Available: https://www.bugcrowd.com
[117] Hackerone. [Online]. Available: https://hackerone.com
[118] 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.
[119] Domain Spoofing in Redirect Page Using RTLO. [Online]. Available: https://hackerone.com/reports/299403
[120] Google DoubleClick Open Redirect. [Online]. Available: https://packetstormsecurity.com/files/129113/
[121] Open Redirect Vulnerability (OBB-2066676). [Online]. Available: https://www.openbugbounty.org/reports/2066676/

#	Pattern ID	Params	Count	New
1	A1	ACTION, action_url, affiliateRedirectURL, away, back_url, backTo, backURL, base, burl, call_url, callback_url, cburl, callbackLocation, came_from, clickurl, continue, ct0, current_page, data, dest, destino, domain, ext, externalRedirect, externalUrl, fail, forward, FORWARD_URL, gHomePage, go, goto, home, hostname, intentUrl, jump, jump_url, layer, link, linkAddress, linkback, lite_url, location, login, login[redirect], login_redirect_url, logout, mgnlReturnTo, net, oadest, old, origin, originUrl, page, pagina, path, post_logout_redirect_uri, previousUrl, promerium_redirect_url, purl, qurl, rd, recurl, redirect, redirect_to, redirect_to, redirect_uri, redirect_url, redirectID, redirectOk, redirectto, redirectUri, redirectUrl, ref, refer, referer, referurl, request, request_uri, RequestedPage, resizewidgeturl, ret_url, RetourUrl, return, return_uri, return_url, returnto, ReturnUrl, reurl, rurl, send, sendTo, service, sp_url, src, st.link, submit-url, success, target, target_link_uri, target_url, TargetURL, to, uri, url, urlRedirect, v, view_url, _next, next	109	59
2	B1	action, callbackLocation, cont, forward, goto, link, loc, location, next, redir, redirect, referurl, return, targetUrl, targetAction, view, callbackUri	17	1
3	B2	advance, callback, callbackUri, ext, fetch, go, goto, redir, ref, return, submitUrl, view, redirect	13	0
4	C1	ajaxUI, action, backTo, backurl, continue, dest, destino, domain, ext, forward, forward_url, gHomePage, go, goto, home, location, next, origin, page, recurl, redirect, redirect_to, redirect_uri, redirect_url, redirectTo, RedirectUrl, referer, return_uri, return_url, returnto, returnUrl, src, to, uri, url	35	35

TABLE VIII: The complete list of parameters for vulnerability indicator patterns in Table III. Parameters marked in cyan color are newly observed, as we did not find them during our review of existing vulnerability reports.

#	Key	Domains	URLs
1	url	102	1224
2	domain	52	766
3	redir	39	891
4	redirect	26	427
5	next	17	245
6	to	14	192
7	r	12	204
8	cburl	9	38
9	redirect_uri	9	135
10	returnto	6	121

TABLE IX: Top 10 URL query parameter keys with the highest number of distinct domains that use the parameter for (open) redirections.

Appendix

A. Additional Evaluation Details

Vulnerability Notification and Vendor Feedback. The open redirect vulnerabilities identified in this paper impact 872 websites, of which for 332 sites we created an exploit. Our notification process began in June 2023, aligning with the discovery of these vulnerabilities, adhering closely to established vulnerability notification best practices [118]. Prioritizing our reports based on known exploits, we sent an initial notification containing the vulnerability description and proof-of-concept exploits, with monthly subsequent reminders. As of the paper's writing, all 332 sites with created exploits have been notified at least once, with 71 confirming the issues and 49 successfully patching them, including popular platforms like TP-Link, Adobe, Starz, and WebNovel. For the remaining open redirects, we needed to contact 540 sites, for which we sought the support of our national CSIRT in January 2024. We observed that site operators are generally reluctant to address open redirections unless they are shown to have broader impacts. At the time of writing this paper, only 18 sites decided patching among the 540 sites where we found no escalations of open redirects.

B. Additional Case Studies

We present additional case studies of the confirmed attacks.

OK. We found an open redirect vulnerability in *ok.ru.* through a query parameter named `st.link`. We exploited this vulnerability to leak OAuth tokens, resulting in user account takeover. OAuth [17] is a mechanism through which users

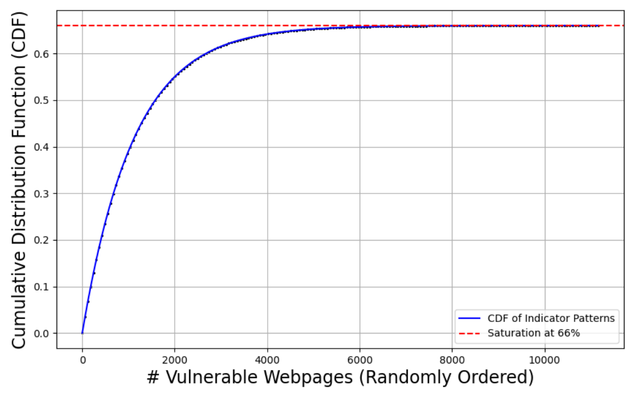
Category	Attack Technique	Example	🗄	◉	References
URI schemes (Basic)	URL (Identity)	evil.com	✗		[2, 3, 28, 30, 31, 69, 74, 87]
	Data	data:text/html;base64,XSS-payload		✗	[76, 87]
	JS	javascript:alert(1)//		✗	[72–74, 76, 87]
Scheme	Use Backslash	https:\\evil.com	✗	✗	[67, 76]
	Escape Slash	https:\/\/evil.com	✗	✗	[76]
	No Slashes	[]evil.com	✗	✗	[87]
	Relative URI	[]//evil.com	✗		[29, 76, 87]
	Encode Colon	https%3A//evil.com	✗		[64, 87]
	Encode Slash	https:%2F%2Fevil.com	✗		[64, 67, 87]
	Encode Specials	https%3A%2F%2Fevil.com	✗		[64, 87]
Netloc	Unicode Dot Encoding	https://evil%E3%80%82com	✗		[87]
	Unicode Normalization	https://evil.c$^{om/s}$.trusted.com	✗		[76]
	Right-to-Left Override	trusted.com@%E2%80%AE@moc.live	✗		[77, 87, 119]
	Prepend Whitelist	trusted.com.evil.com	✗	✗	[76, 77, 87]
	Prepend Authentication	trusted.com@evil.com	✗		[67, 76, 77, 87]
Path	Directory Confusion	evil.com/path/www.trusted.com	✗		[76, 77]
Query	Parameter Pollution	redir=trusted.com&redir=evil.com	✗		[76, 87]
IP	Decimal	1.2.3.4	✗		[2, 66, 87]
	Dotless Decimal	16909060	✗		[66, 87]
	Hex	0x01.0x02.0x03.0x04	✗		[66]
	Dotless Hex	0x01020304	✗		[66]
	Octal	0001.0002.0003.0004	✗		[66]
	Dotless Octal	0x01020304	✗		[66]
Injection	Null Byte	evil%00.com	✗		[76, 77]
	CRLF	java%0d%0ascript%0d%0a:alert(1)	✗		[65, 72, 76, 77]
Other	Alternating Caps	jAvAsCrIpT:alert(1)		✗	[72]

TABLE X: Overview of open redirect attack techniques. The examples redirect `trusted.com` to `evil.com` with an IP of `1.2.3.4`. The column 🗄 marks techniques that may bypass server-side filters whereas ◉ shows techniques bypassing client-side input validation checks. The ✗ symbol marks applicable attack techniques.

		Candidate		🐞 Vuln.	
Source	**Pattern**	**URLs**	**Sites**	**URLs**	**Sites**
Internet Archive	A1	162,562	6,108	205	171
	A2	15,675	1,270	44	37
	B1	8,445	965	12	8
	B2	1,502	417	3	1
	B3	198	44	1	1
	B4	21	5	0	0
	Total	188,403	8,045	**265**	**218**
Google Search	A1	661	371	12	11
	A2	380	123	7	7
	B1	121	56	2	2
	B2	49	12	0	0
	B3	17	5	0	0
	B4	9	2	1	1
	Total	1,237	569	**22**	**21**
Live Crawl	A1	19,210	2,045	40	37
	A2	2,866	706	15	15
	B1	404	210	4	4
	B2	18	10	0	0
	B3	2	4	1	1
	B4	1	1	0	0
	C1	2,786	828	24	23
	C2	655	399	8	6
	C3	223	155	2	2
	Total	26,163	3,089	**94**	**88**
Total		214,645	8,045	**375**	**326**

TABLE XI: Overview of vulnerability mining results.

Fig. 3: Growth of indicator patterns across vulnerable webpages in P_1 dataset.



can provide service providers with access tokens for specific scopes via an identity provider. We found that *ok.ru* allows its users to authenticate via *mail.ru* identity provider which works with a `redirect_uri` parameter (see, e.g., [17, 74, 88]). Once the user grants the requested permissions to the service provider, the identity provider sends an authorization code to the specified destination in the `redirect_uri`. However, *mail.ru* checks that the `redirect_uri` value belong to the *ok.ru* site before sending the authorizaiton code. To bypass this check, we can exploit the open redirect vulnerability in `ok.ru` to chain the redirects and forward the request to an attacker-controlled domain, stealing the authorization code left in the request `referrer` HTTP header.

Listing 4: Excerpt of a client-side open redirect vulnerability in *tp-link.com* escalated to DOM XSS.

```
1 let $url = new URLSearchParams(location.search).get('url');
2 let $params = location.hash.slice(1).toLowerCase();
3 let $product = params.match('&pview=true');
4 if($product && screen.width<=1024){
5     // $url: javascript:alert(1);
6     location.href=$url;}
```

Listing 5: Excerpt of a client-side open redirect vulnerability in *udn.com* escalated to DOM XSS.

```
1 var toUrl = document.URL;
2 var whereIs = toUrl.indexOf("redir=");
3 whereIs_end = toUrl.indexOf("&site=");
4 if ( whereIs_end == -1  || whereIs_end < whereIs ){
5     whereIs_end=toUrl.length; }
6 url=toUrl.substring(whereIs+6,whereIs_end);
7 setTimeout("window.location=url",1000);
```

Google WebLight and DoubleClick. We found a vulnerabilitiy in *weblight* where a URL query parameter `u` was not validated and allowed redirection to arbitrary domains. Similarly, we found an open redirection in the DoubleClick advertising service where the destination of a query parameter `adurl` was not validated before redirection. However, while confirming the open redirection in DoubleClick, we realized that it is a known issue (see, e.g., [120, 121]) which Google decided not to patch.

TP-Link and UDN. Listing 4 shows a simplified version of an inline script vulnerable to client-side open redirect vulnerability that we found in *tp-link.com*. The vulnerability affects the product view functionality of the store. The program initially extracts the value of the `url` query parameter in line 1, examines the necessity of the product view functionality based on a flag in the URL fragment (lines 3-4), and subsequently employs it as the value for `location.href` in line 6. However, a notable input validation vulnerability exists as the code does not sanitize the `url` value against `javascript:` URIs, making it possible to acheive DOM XSS attacks.

A similar vulnerability affects `udn.com`, as illustrated in Listing 5. Here, the code reads the value of the query parameter `redir` in line 2 and subsequently sets the value of window location to the read value after one seconds using the `setTimeout` instruction (line 7). Similarly to TP-Link, it can be exploited for arbitrary client-side code execution.

C. Mitigating Escalations

In this section, we examine cases where we were unable to escalate a confirmed client-side open redirect vulnerability to XSS. Our goal is to identify potential mitigation strategies to counteract escalations. To do that, we manually analyzed all *JavaScript-based* open redirections we found via vulnerability mining in §V-B. Out of the 375 open redirect URLs, 202 are JavaScript-based, and among them, 108 cannot be exploited for XSS attack. By examining these 108 URLs, we found two mitigating reasons. First, CSP can mitigate the escalation of client-side redirections to XSS attacks by disallowing `javascript:` URIs. We found that CSP blocked the majority of the XSS escalations (i.e., 83 out of 108 or 76%), whereas the remaining webpages (24%) had proper input sanitization procedures that stopped the injection of inline JavaScript code.

D. URL De-duplication

In §V-A, we performed URL de-duplication to limit our results to unique endpoints only. To do that, we used a heuristic-based approach to detect dynamic URL components (e.g., numbers in the path) by comparing the syntax and value of discovered URL elements with one another. We classified a URL component as dynamic if the prefix and the suffix (if present) of the component remained constant, while the value of the component changed in candidate URLs. We observed that this heuristic is conservative, as its application, following manual review of 1000 random URL groupings, revealed little-to-no false positives in our dataset (rate of 1/1000). In summary, we considered the following types: (i) `NUMBER` for all strings that represent a number; (ii) `RESOURCE` for all strings that contain a dot and the latter part is a valid MIME type (e.g., "image.png"); (iii) `SIMPLE` for all strings that only contain alphanumeric characters; (iv) `COMPOSITE` for every string that contains a dash or underscore (e.g., "blog-title"); and (v) `COMPLEX` for every string not meeting any of the criteria before (e.g., "%46%4F%4F"). If multiple predicates match, we choose the type corresponding to the first match.