Scripted Henchmen: Leveraging XS-Leaks for Cross-Site Vulnerability Detection

Tom Van Goethem[†], Iskander Sanchez-Rola[*] and Wouter Joosen[†]
[†]imec-DistriNet, KU Leuven
[*]Norton Research Group

***Abstract*—The key security principle that browsers adhere to, such as the same-origin policy and site isolation, ensure that when visiting a potentially untrusted website, the web page is loaded in an isolated environment. These security measures aim to prevent a malicious site from extracting information about cross-origin resources. However, in recent years, several techniques have been discovered that leak potentially sensitive information from responses sent by other sites. In this paper, we show that these XS-Leaks can be used to force an unwitting visitor to detect prevalent web vulnerabilities in other websites during a visit to a malicious web page. This lets an adversary leverage the computing and network resources of visitors and send malicious requests from a large variety of trustworthy IP addresses originating from residential networks. Finally, we find that currently deployed security measures are inadequate to thwart the realistic threat of cross-origin vulnerability detection.**

I. Introduction

As we visit a website, we usually do not know which tasks our browser is performing in the background. This characteristic has previously been abused to covertly perform cryptomining on the browser of unwitting visitors [30], [6], [9]. Similarly, malicious websites might determine whether a browser is susceptible to known vulnerabilities and launch drive-by download attacks [43], [54], [15]. Although both crypojacking and drive-by download attacks have diminished, either because it is no longer appealing from a financial point of view [71], or because browsers have become increasingly hardened [18], in this paper we show that the threat of illicit background operations while visiting a website is still present.

Concretely, we explore in depth how a malicious JavaScript payload served to unwitting visitors can force the browser to detect vulnerabilities in other websites and report them back to the adversary. In essence, the attacker abuses the resources (computing and network) of a user who is lured to a malicious website to launch cross-site attacks against targeted websites. The attacks can be used to determine whether certain websites are susceptible to prominent web vulnerabilities such as cross-site scripting (XSS) and SQL injection. This provides the adversary with several advantages. First, the IP addresses from which the vulnerability-detection attempts originate, come from residential networks and thus are more likely to be trusted by intermediary security services, e.g. CDNs. Second, the IP addresses are changing with every new visitor, allowing the attacker to circumvent IP-based rate limiting systems that e.g. protect the authentication flow. Finally, according to our estimates and depending on the number of visitors that can be lured to the malicious website, the attacker might have to pay up to thousands of dollars to achieve the same characteristics and attack capacities with existing services.

Because of the same-origin policy, a key browser security principle, it is not possible to directly access cross-origin responses, and hence various techniques are required to circumvent these protections. We explore the feasibility of using three different methods: abusing specific configurations of the cross-origin resource sharing mechanism, leveraging web rehosting services, and finally using XS-Leaks to leak metadata of cross-origin responses. As the two former methods provide the adversary with access to the full response body, attacks can be launched in a similar way as to how an adversary would from their own server. The methods that rely on leaking side-channel information about the responses make use of a combination of known techniques and novel XS-Leaks that we discovered as part of our research.

Although websites are able to defend themselves from the threat of cross-origin vulnerability detection by deploying various security features, we find that in reality, only a tiny fraction of sites is adequately protected. Furthermore, to fully counter these attacks, an effort from various entities within the web ecosystem is required: web rehosting services should adopt effective defenses, website operators should deploy adequate configurations, and browser vendors should mitigate cross-origin leaks and, ideally, enable security measures by default. As the complete mitigation of the cross-origin vulnerability detection methods is an arduous process, we believe that for the time being the attacks pose a realistic threat to the web ecosystem, which should be taken into consideration by websites and third-party security services.

Our main contributions can be summarized as follows:

- We thoroughly analyze the threat posed by cross-origin vulnerability detection attacks based on three general methods: abusing lax CORS policies, leveraging web rehosting services, and using XS-Leaks to infer side-channel information.
- We uncover two novel XS-Leak techniques that enable cross-site vulnerability detection. These leaks can also be applied in the more traditional context of cross-site attacks.
- We develop a testbed to validate the effectiveness of the attacks, and, using extensive public datasets, we evaluate their performance and estimate the costs to achieve

equivalent attack characteristics with existing services.
- We evaluate how currently deployed defenses, both by websites and browser engines, mitigate the various vulnerability detection techniques and propose suggestions to improve the current state.

II. Motivation & threat model

Despite the continuous efforts of the security community to thwart web attacks and cybercrime in general, the number of attacks launched against web applications is still increasing. In their latest "State of the Internet" report, Akamai finds that the number of daily attacks has significantly grown in 2021, with a peak of 113M attacks detected in a single day in June 2021, three times the number of attacks observed the previous year [1]. Similarly, Verizon reports in their latest Data Breach Investigations Report (DBIR) that basic web application attacks are increasingly more prominent as the cause of security incidents or data breaches over the past several years [72]. Detecting these vulnerabilities at large scale may require extensive computing and bandwidth resources, and their costs may outweigh any potential profits, e.g. from selling data leaked from compromised websites. In this paper we analyze a new threat where the resources of web visitors are used instead.

A. Threat model

We explore how various techniques can be leveraged to circumvent the same-origin policy and use the leaked information, either the full response or side-channel data, to detect common vulnerabilities in websites. More concretely, when an unwitting user visits a malicious web page, which could either be a website that is purposefully set up, compromised, or that contains a malicious advertisement, a JavaScript payload will force the user's browser to initiate (cross-origin) requests to other websites in order to detect whether these contain vulnerabilities. In essence, the malicious JavaScript payload will operate as a vulnerability scanner: by analyzing the (metadata of) responses to attacker-initiated requests, it is possible to infer whether a vulnerability such as XSS or SQL injection is present. It is important to note that the requests to the targeted website originate from the IP address of the user who is visiting the malicious web page. Finally, in our threat model, we assume that the attacker can force the user's browser to make illicit requests for the duration of a web page visit, which is on average 54 seconds according to a recent report by Contentsquare [14].

B. Advantages

Compared to a typical *direct* attack where the illicit requests originate from the attacker-operated server, cross-origin vulnerability detection methods presented in this paper have several benefits that allow an adversary to overcome defensive measures. Of course, these attack methods also have limitations, as they rely third-party services or leverage side-channel information. We discuss these limitations in detail in Section VI-B. Cross-origin vulnerability detection has the following advantages compared to direct attacks:

1) IP variability. As the vulnerability detection requests originate from the IP address of a visitor, a new IP address will be used for every new visitor. This high variability of IP addresses can be valuable for an adversary, as websites may deploy rate limiting systems. In a direct attack setting this may be more challenging to achieve as the attacker would have to pay for additional IP addresses in a cloud environment (on AWS: $0.10 USD per IP remap [2]), or cycle through IP addresses of a VPN service, which may also be limited.

2) IP reputation. In addition to the continuously changing IP addresses, another benefit is that these IPs are of real users, originating from residential networks. As many websites deploy CDN services to thwart automated bots while still allowing organic traffic, the IP reputation that users have built up during prior browsing sessions is likely to have a favorable effect on the bot-detection mechanisms.

To evaluate the extent to which websites block traffic based on the reputation of the IP address, we performed an experiment where we visited the home page of the top 1,000 Tranco domains [33] via our university network, as well as via Tor, which is known to be a common source of malicious traffic (and the IPs are thus more likely to have a bad reputation). For each page visit we took a screenshot and then manually labeled these to determine whether the website blocked the visit, e.g. by requiring a CAPTCHA or showing an "Access Denied" message. We excluded page visits that did not load successfully, either due to a timeout or because the domain did not host a valid website. We found that 15.53% of websites that were visited over Tor blocked the initial visit to the homepage. Interestingly, 5.62% websites blocked access from our university's network (although half of those still allowed Tor traffic); as the page visits were sequential, the CDNs serving those websites probably detected this automated traffic.

Another indication of the importance of IP reputation is the rise of residential IP proxies [40]. Although these services provide similar benefits, these come at a significant financial cost, with a fixed subscription price costing several hundreds USD along with a usage price of $10-15 USD per GB [8].

3) Operational costs. One of the key benefits of cross-origin vulnerability detection is that the operational cost is minimal. The adversary only needs to host a malicious web page, which can be done for free via a public service such as GitHub pages. The performance of the vulnerability detection methods mostly depends on the number of visitors that are driven to the malicious web page. The attacker can drive visitors to their web page using a variety of techniques, most of which are low-cost. Such techniques include blackhat SEO and search poisoning [37], [74], [16], [67], spamming social networks [21], [63], email and instant messaging spam [27], [59], and advertisements [35], [64]. Additionally, when a vulnerability was detected and exploited, the attacker could include the malicious payload in the compromised website [52], [36] and benefit from its organic traffic.

III. Accessing cross-origin responses

In a typical direct web attack, the adversary will send a series of requests to the targeted server and will determine whether a vulnerability is present based on the responses that are returned. For example, to detect whether the website is susceptible to a reflected XSS attack, the adversary can include a JavaScript payload in one of the request query parameters and subsequently parse the response to determine whether the payload was adequately sanitized or escaped. Similarly, to test for SQL injection attacks, the adversary will determine whether a part of the payload was executed in the SQL query by analyzing the response that was returned (e.g. the number of results, returning an empty page, ...). Because of the same-origin policy, access to the cross-origin response body is blocked by default. In this section we explore two techniques through which this restriction can be circumvented, and thus allows having complete access to the response body.

A. Cross-origin resource sharing

One of the ways that browsers allow a relaxation of the same-origin policy is through the cross-origin resource sharing (CORS) mechanism [69]. By defining certain response headers, a website can opt in to provide other origins access to certain resources. More concretely, the `Access-Control-Allow-Origin` (ACAO) header can be used to indicate which origin is allowed access to the response. This header also allows the wildcard value `*`, which gives every origin access to the response. A caveat with this wildcard value is that it does not support credentialed requests, i.e. requests that include cookies. Therefore only vulnerabilities in the unauthenticated part of a website can be detected with this method.

To determine the prevalence of this wildcard CORS header, we queried the latest (December 2022) HTTP Archive crawl, which has detailed information about the homepage of more than 12.5M sites. We found a total of 265,232 websites (2.11%) for which the homepage was served with the ACAO header set to `*`. Interestingly, we find that this header is more commonly present on websites that are more frequently visited: of the top 1,000 websites (according the Chrome User Experience Report [23]) 9.09% allow all sites to access the cross-origin response.

Listing 1: Example vulnerability detection payload.

```
let payload = '<script src="//atk.com/"></script>';
let url = `https://target.com/?param=${payload}`;
let options = {mode: "cors", credentials: "omit"};
let response = await fetch(url, options);
let body = await response.text();
let parser = new DOMParser();
let doc = parser.parseFromString(body, 'text/html');
let el = doc.querySelector('script[src="//atk.com/"]')
if (el) {
  // script was injected; vulnerability detected
  reportVuln(payload)
}
```

An example of how a reflected XSS vulnerability can be detected on targeted website that sets a wildcard ACAO header is shown in Listing 1. This script will request an endpoint of the website where one of the URL parameters is set to a payload that would include a script from `atk.com`. Next it will use the DOMParser API [38] to parse the returned HTML code and uses a CSS selector to determine whether the script was parsed as such. This would indicate that the query parameter was not adequately sanitized when it was reflected in the response, and thus an XSS vulnerability was detected.

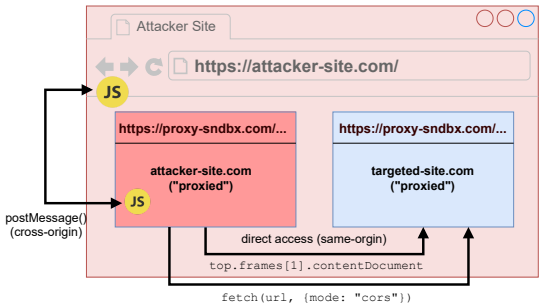

Fig. 1: Abusing web rehosting services to circumvent SOP.

B. Web rehosting services

1) Overview: Web rehosting services, which consist of online web proxies, web page translators and archival services, take the content of a provided web page and serve it under their own (sub)domain. In essence, these services operate as an open reverse proxy, and thus will forward requests to the targeted website, modify it (e.g. translate the text or inject advertisements), and send the response back to the user. When all websites that are accessed through the service share the same origin, i.e. the origin of the rehosting service, this can be leveraged to circumvent the same-origin policy. In prior work, Watanabe et al. explored the consequences of this lack of origin isolation, and described several attacks such as a persistent MITM using service workers, and abusing privileges (e.g. camera, microphone) that were provided to other rehosted sites [75]. In their work, Watanabe et al. focus on the security and privacy consequences for the users of these web rehosting services, where the adversary is mainly interested in leaking information from prior visits to other rehosted sites. Next, we show how the lack of origin isolation in most of these services also allows us to perform cross-origin vulnerability detection.

An overview of how the SOP circumvention by leveraging web rehosting services is achieved, is shown in Figure 1. When a user visits an attacker-controlled website, this site will include an iframe of the rehosting service that "rehosts" an attacker-controlled web page. In the figure, this is `attacker-site.com` that is rehosted on a sandboxed origin of the proxy service: `https://proxy-sndbx.com`. From this origin, the attacker has access to any other resource that is served through the rehosting service. The attacker could opt to access other rehosted resources either directly by using a same-origin `fetch()`, or letting the browser render the rehosted web pages in another iframe. The latter can still be accessed directly via the DOM as the rehosted attacker page has the same origin as the rehosted target website. Accessing content from the iframe directly can be advantageous to detect vulnerabilities such as DOM-based XSS, for which the web

TABLE I: Summary of the susceptibility of web rehosting services to cross-origin vulnerability detection.

Category	Service	allows SOP bypass	Defenses: framing protection	Defenses: removes scripts/iframes	Defenses: cookie verification	Defenses: CSRF token	Defenses: custom subdomain	all defenses can be circumvented	can install SW	supports cookies
Proxy	ProxySite	●	○	○	●	○	○	●	●	●
	Hide My Ass!	●	●	●	○	○	○	●	●	●
	Sitenable Web Proxy	●	○	○	○	○	○	●	●	○
	FilterBypass	○	●	●	●	●	○	○	○	●
	ProxFree	●	○	○	●	○	○	●	●	●
	hidester	●	○	○	●	○	○	●	●	●
	GenMirror	●	○	●	●	○	○	●	●	●
Translator	Google Translate	○	●	○	○	○	●	○	○	○
	Weblio	●	○	○	○	○	○	●	●	○
	Yandex.Translate	○	○	○	○	●	○	○	●	○
	Baidu Translate	●	○	○	○		○	●	○	○
Archive	Wayback Machine	●	○	○	○	○	○	●	○	○
	Google Cache	●	○	○	○	○	○	●	○	○
	FreezePage	●	○	●	○	○	○	●	○	○
	Total	11/14	3/14	3/14	5/14	2/14	1/14	11/14	8/14	6/14

page needs to be rendered. When the web rehosting service forwards the `Set-Cookie` headers of the rehosted web page, it is also possible to perform the vulnerability detection on authenticated parts of websites.

2) Evaluation: To evaluate the extent to which these web rehosting services can be abused to launch cross-site vulnerability detection attacks, we analyzed the set of rehosting services that were previously evaluated by Watanabe et al. [75] and were still available at the time of our analysis (January 2023). The results of our analysis are shown in Table I. We find that 12 out of the 14 analyzed web rehosting services can be abused to circumvent the same-origin policy in order to perform cross-site vulnerability detection. Compared to the study of Watanabe et al. [75], we find that only Google Translate took adequate efforts to prevent attacks.

3) Bypassing controls: Although two services, GenMirror and FreezePage, were considered not exploitable in prior work, as these block JavaScript on the page, we managed to bypass all imposed defenses, and find that these are in fact vulnerable. GenMirror blocks JavaScript responses based on the response content type, and would thus prevent an attacker-controlled JavaScript payload to be executed. However, this can be trivially bypassed by using capitalization in the content type (e.g. `text/jaVascript`), which was missed by the service but is supported by the browser. The FreezePage proxy service rewrites HTML to exclude any `<script>` elements. We managed to circumvent this defense by using `<iframe>` elements with a `srcdoc` attribute and subsequently using HTML entities to encode the script element: `&lt;script&gt;` [51].

Other defensive mechanisms that were present include framing protection, verifying that the user has a valid cookie, and enforcing that page visits can only be made through a form for which a CSRF token is required. For the web rehosting services that do not prevent installing a service worker, we could circumvent the framing protection defense by first installing the service worker and subsequently letting it intercept all requests. The service worker would then act as a MITM and remove the `X-Frame-Options` and CSP headers, disabling the framing protection.

Because several browsers set the `SameSite` attribute of cookies to `Lax` by default, these cookies will not be attached to the request when the rehosting service is included in a cross-origin iframe. Consequently, services that require the presence of a valid cookie can only be used in a top-level document (i.e. when the rehosting service is not included in an iframe). However, we found that for all services except FilterBypass this restriction could be lifted by setting a cookie via JavaScript with the `SameSite` attribute set to `None`. By giving this cookie a stricter `Path` or `Domain` attribute, it will override the default cookie, and will be included in all subsequent requests.

Finally, we evaluated whether it was possible to install a service worker that could intercept requests to other rehosted sites. For this, the service had to proxy JavaScript files, and leave them unmodified. Furthermore, the origin and path on which the rehosted site is hosted should be the same for all sites. On 8 rehosting services we found that a service worker could be installed. This could be leveraged to extend the lifetime during which attacks can be executed (service workers remain active for some time after the user has left a page [13]). Furthermore, any subsequent time that the user would visit the rehosting service, the attacker could inject a malicious payload and make the user's browser perform cross-origin vulnerability detection.

Summary. We found that only two defenses could not be circumvented: using custom subdomains and limiting the user's actions by using a form that requires a valid CSRF token. As such, we believe that these are the most effective methods to prevent these services from being abused to circumvent the same-origin policy. We notified the affected web rehosting services of our findings.

IV. Leveraging side-channel information

In the previous section we explored how web technologies could be leveraged to access cross-site response bodies. A downside, from the attacker's perspective, of these methods is that these either rely on a specific configuration of the target server (CORS headers), or require the support of third-party web rehosting services. In this section we focus on techniques

that are more generic and that can be used to detect prevalent web vulnerabilities in a cross-origin context without requiring opt-in mechanisms or third-party services. These techniques leverage side-channel attacks, commonly referred to as XS-Leaks, to leak meta-information about cross-origin responses. XS-Leaks have been commonly used to detect information about the response to *authenticated* requests, i.e. to infer the state that the victim shares with a targeted websites. However, in the context of cross-site vulnerability detection we aim to determine the state of the targeted website instead, i.e. to determine whether a vulnerability is present.

A. Overview of known XS-Leaks

Next, we provide a brief summary of the known XS-Leak techniques that we leverage for the purpose of cross-origin vulnerability detection. For an extensive overview of all XS-Leaks that have been discovered to date, we refer to the XS-Leaks Wiki [81] and the research by Sudhodanan et al. [61], Knittel et al. [28], and Van Goethem et al. [65].

Response size. In 2007, Bortz and Boneh showed that it is possible to leverage the network response time to estimate the size of a resource [7]. As these timing measurements are subject to the network conditions of the client, attacks may be impractical. More recently, Gelernter and Herzberg showed that by leveraging response inflation, these techniques could still be successfully exploited [19]. Moreover, Van Goethem et al. introduced several mechanisms to estimate the response size based on the time it takes the browser to parse or process its contents, thereby making the attack independent of the client's network conditions [66].

Response status. Several researchers have shown that it is possible to differentiate between different response statuses, e.g. 200 vs. 500 [34], [57]. Some other techniques can be used to detect whether a redirect has occurred. For example the History API leaks the number of redirects that occurred via the `history.length` property [80]. This however only applies to client-side redirects, i.e. via JavaScript or a `<meta>` tag. To detect server-side redirects, i.e. responses with a 30x status code, it is possible to leverage the maximum number of redirects that is allowed before a network error is returned [24].

Resource content and operations. Another aspect that may be detectable is based on the content of a HTML-resource and the operations it performs. For instance, when an HTML-resource is included in an iframe, or opened via `window.open()`, it is possible to determine the number of iframes it contains by accessing the `frames.length` property. Similarly, a resource may perform an action that can be observed from a cross-origin window. Sudhodanan et al. found that certain web pages would send a `postMessage()` message based on the state of the user [61].

B. Novel XS-Leaks

As part of our research, we uncovered two novel XS-Leak attacks. Although we use these techniques as part of our vulnerability detection methods, they can also be used to leak information about the state a user has with a targeted website.

Presence of subresources. When a web page is rendered, the same-origin policy prevents any cross-origin window from inferring whether any subresources were fetched. We found that this protection could be circumvented by carefully arranging the navigation of an iframe's browsing context. More concretely, an adversary will first create a new iframe, which will have the same origin as the embedding web page, thus allowing access to its DOM properties (e.g. `iframe.contentWindow.origin`). Next, the `src` attribute will be set to the targeted web page. This will trigger the browser's renderer to fetch the HTML document. Once the document has been downloaded, the iframe's browsing context will be navigated to the target page, and thus the origin of the iframe's document will change, making it no longer accessible by the embedding attacker page. It can be observed by the adversary when this navigation happens, as it coincides with the time that accessing a DOM property of the iframe will throw an exception.

As soon as the attacker catches an exception, it will set the `src` attribute back to `about:blank`. Although this empty document does not take a long time to load, the renderer at this time is still occupied with parsing the HTML from the target page. We found that if this page does not contain any subresources, the browser would first fire a `load` event on the iframe before the navigation to `about:blank` occurs. If subresources need to be downloaded, the `load` event would not be fired. We tested this in all major browser engines, and found that it consistently revealed the presence or absence of subresources. Finally, we found that this method could also be used to detect the presence of framing protection (via XFO or CSP), as the network error page does not include any remote resources. A detailed timeline of this technique can be found in Figure 4 in the Appendix.

CDN cache status. For performance reasons, many websites serve resources via a CDN, which has a world-wide network of "points of presence" (POP) that deliver cached resources to users in their geographic vicinity. To facilitate debugging, most CDNs use a custom header to indicate whether a resource was served from the cache or had to be obtained from the origin server (a recent effort aims to standardize such a header [44]). In the case when the cache state of a specific resource is directly related to a secret, i.e. unknown to the adversary, condition, determining this cache state reveals this secret. We find that for most CDNs, the cache state could be determined based on the response headers, as was previously shown by Mirheidari et al [41], [42]. In a cross-site attack, this would require the CDN to set an `Access-Control-Expose-Headers` (ACEH) response header along with a permissive `Access-Control-Allow-Origin` (ACAO) header, which we found in over 44,000 instances (out of 42.5M requests). Alternatively, we found that the timeless timing attacks introduced by Van Goethem et al. [68] could also be used to reliably determine the CDN cache status of a resource.

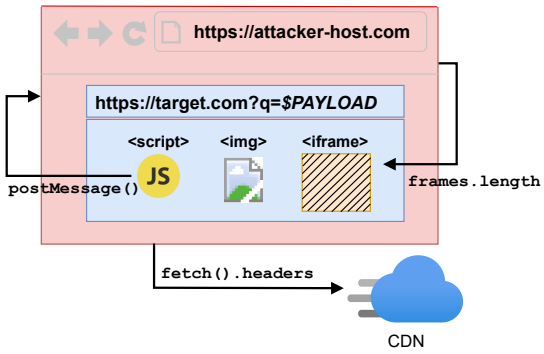

Fig. 2: Cross-site XSS detection.

V. Vulnerability detection through XS-Leaks

In this section, we explore in detail how XS-Leaks can be used to detect the most common web vulnerabilities, based on the OWASP top 10 [46] and reports on the prevalence of real-world web attacks [1], [72]. We report on how four prominent web vulnerabilities can be detected in a cross-origin context, using the computing resources and network connection of unwitting visitors.

A. Cross-site scripting

Caused by improperly escaping or encoding attacker-controlled input, XSS vulnerabilities allow the adversary to execute JavaScript code in the origin of the targeted website.

postMessage. To detect the presence of an XSS vulnerability, the attacker can try to inject a JavaScript payload that will perform cross-origin communication. The most straightforward way to do this is by using the postMessage API. Hence, when a message is received, the adversary can infer that the payload was successfully executed, implying that a vulnerability is present. In practice, the attacker can choose to include the targeted web page either as an iframe or in a new top-level window. An advantage of the former is that the detection attempts remain invisible from the visitor. However, the iframe-based method only works if the targeted page does not enforce framing protection.

Frames. Instead of injecting JavaScript code, the adversary could also try to inject other elements as this will still reveal whether the attacker-provided content is not properly sanitized or encoded. By injecting a number of `<iframe>` elements, the attacker could easily determine whether this injection was successful by simply counting the number of frames - a property that is accessible in a cross-site context due to historical reasons.

Prerender. Finally, a third technique relies on the prerender functionality which, at the time of writing, is only available in Chromium-based browsers [5]. Despite what the name suggests, prerender will only prefetch the subresources (excluding iframes) of a web page. Furthermore, the prerender process occurs in the background and the page requesting the prerender (using a `<link>` element) has no reference to it. As a result, neither of the previous two techniques can be used to detect improper sanitization of the attacker payload. Instead, we can leverage our novel "CDN cache status" leak. Concretely, the attacker's payload can attempt to include an `<img>` element with the `src` attribute set to one of the hosts that set its ACAO and ACEH headers to reveal the cache status. Note that although we found that only 0.10% of resources have such headers, only a single host with such a configuration is required for the purpose of the attack. With over 44,000 instances that match the criteria, an attacker has plenty of options.

B. SQL injection

We found several XS-Leak methods that can be leveraged to detect SQL injection vulnerabilities in a cross-origin context. In essence, the adversary will try to determine whether their payload was able to control the SQL query, which can be revealed by returning additional data (response-based), executing a function that takes a long time (timing-based), or that would throw an error (error-based). We describe how each of these three methods can be detected using XS-Leaks.

Response inflation. If the attacker is able to perform an injection where an additional query is included in the result (e.g. by using an `UNION SELECT` construct), it is possible to inflate the response document. More precisely, the attacker can trigger the server to return a long random string, and measure the response time. Note that because most web servers use gzip compression for responses, the string should have a low compression ratio. In our experiments, we found that using a combination of `REPEAT("a", 1e6)` and the `DES_ENCRYPT()` function can inflate the response with approximately 1MB, which can be easily detected using a timing attack.

Time inflation. When launching a timing-based attack, the attacker will trigger a delay in the execution (e.g. with the `SLEEP()` function) when the exploitation is successful. If a sufficiently large delay is triggered, the attacker can easily observe this with a network timing attack [7], or use a timeless timing attack to measure the difference in execution time on the server [68].

Error detection. Finally, to detect SQL injection attacks via the error-based method [22], we can leverage any XS-Leak that reveals the error status code of the response [34], [81]. Note that if the server would catch the error caused by the SQL query and simply display a message, it would not be possible to reliably determine this difference using XS-Leaks. Alternatively, the "presence of subresources" XS-Leak described in Section IV-B can be used as well, assuming the error page does not contain subresources.

C. Login brute forcing

A common cause of compromise on the Web is using a weak password (guessed in a brute force attack), or reusing the same password for different websites. In order to limit the success of such attacks, websites often rate limit the login flow. This makes it a particularly interesting target for cross-site vulnerability detection due to the high variability of IP addresses from which the attacks are launched. To verify the feasibility

of a cross-origin login brute force attack, we analyzed the five most popular content management systems (according to W3Techs these are: WordPress, Shopify, Squarspace, Joomla, and Drupal [73]). Three of these CMSes protect against login-CSRF attacks by requiring a unique token in the login form (Joomla and Squarespace), or by checking for the presence of a `SameSite` cookie (Shopify), and thus are unaffected by the attacks. Nevertheless, we found that WordPress and Drupal, which have an accumulated market share of 67.27% and are installed on 44.6% of websites [73], can still be attacked.

History API. The cross-site login brute force attack can be performed against two operations, either the login mechanism itself or the state-change that happens as a result. In many login mechanisms, including that of WordPress, the user is redirected to the admin panel after successful authentication. By detecting this redirect, the attacker can infer whether the credentials are valid. One way to detect this is by leveraging a side-channel based on `history.length [80]`. More precisely, the attacker first creates a (same-origin) iframe with a form containing the tested credentials as input fields, and the `action` attribute set to the targeted website. Before submitting the form, the attacker records the `history.length` property. Once the form is submitted, a login attempt will be made and eventually the `load` event will fire on the iframe, indicating that the page has loaded. At this point, the attacker will set the `src` attribute of the iframe to a page they control, and read out the `history.length` property. If the login was unsuccessful, and no redirect happened, `history.length` will not have increased. Otherwise, if the credentials were correct, and the client was redirected to `wp-admin/`, in which case the `history.length` property was incremented.

D. Server-side request forgery

In a server-side request forgery (SSRF) attack the adversary tricks the targeted web application to send an attacker-controlled request. Typically this request targets an endpoint that is only accessible locally from the web server, and thus may not be as well-protected as the internet-facing applications. To detect the presence of an SSRF vulnerability, the attacker needs to detect whether the affected endpoint caused a request to be sent.

Web hooks debugger. The adversary can leverage popular third-party services that are used to debug web hooks by providing a unique endpoint where all requests are logged. Examples of such services include RequestBin [50], Beeceptor [4], Webhook.site [76], and several others. Using one of these endpoints as the target URL will log information about the request, if one is made. However, the attacking page can not access this information in a cross-origin context, so either a web rehosting service should be used to circumvent the same-origin policy, or the adversary could access the service directly. Any request logged by the web hook services likely indicates a detected vulnerability.

TABLE II: Runtime of vulnerability detection techniques.

Technique	Execution time	Concurrent?
XSS-IFRAME	RENDER	●
XSS-WINDOW	RENDER_{p95}	○
XSS-PRERENDER	RENDER_{p95} + FETCH	●
SQLi-RESP-INFL	FETCH	●
SQLi-TIME-INFL	FETCH	●
SQLi-ERROR	FETCH	●
LOGIN-HISTORY	2×RENDER	●
SSRF-WEB-HOOK	FETCH + RENDER_proxy	●

E. Evaluation

1) Effectiveness: To evaluate the effectiveness of the different cross-site vulnerability detection mechanisms, we created a testbed with a website based on WordPress (the most popular CMS), in which we introduced several vulnerabilities. We implemented each technique and tested them on the most popular web browsers (Chrome, Firefox, Edge, Safari) [60]. All the detection techniques rely on side-channel information, and thus may be affected by the conditions of the client's environment. To evaluate this impact, we performed experiments under varying conditions (cabled network vs. Wi-Fi connection, interaction from the user on the malicious site, or video streaming in another tab). We found that each technique was highly reliable and consistent, as most either rely on deterministic behavior (e.g. counting the number of frames), or have a very high signal-to-noise ratio (e.g. using a timing side channel to detect a response size difference of 1MB – prior work has shown that a difference as small as 5kB can be reliably detected [66]). Similarly, an additional timing delay of 5 seconds outweighs the jitter on a network connection by several orders of magnitude.

2) Performance: The rate at which the vulnerability detection techniques can be performed mostly depends on the environment of the visitor who is executing the malicious scripts. As most techniques rely on sending requests and observing their side-effects, the latency of the visitor's network connection is typically the limiting factor. For the methods that require rendering a web page, the specifications of the user's machine will also play a role. To evaluate the performance of the different techniques, we determine which operations are required to perform a single detection. In essence, there are two main operations that need to be considered: fetching a resource (FETCH) and rendering a page (RENDER).

In Table II we show the operations that are required to perform a single detection for each of the techniques. For most techniques only FETCH operations are needed. For the detection techniques that rely on rendering a web page in a separate window, the attacker cannot determine when the page finished loading. Hence, we assume that the attacker needs to wait for a duration that matches the 95^{th} percentile of average loading times of web pages. Although vulnerabilities in web pages that load very slowly will not be detected, we believe this would be an acceptable trade-off for the adversary. We note that although the SQLi-ERROR technique includes the target page in an iframe, it will stop loading the page as soon

as the HTML of the page has been downloaded. As a result, only a FETCH operation is required.

To determine the average duration for the different operations we use the measurement data from the Chrome User Experience Report (CRuX) which reflects the loading times for a large number of users for 8M websites [20]. To estimate the duration of the FETCH operation we use the time-to-first-byte (TTFB) measurement, for which we find a median value of 400ms for desktop and 500ms for mobile. For the RENDER operation, we use the time until the `load` event is fired, for which the median values are 1.9s and 2.6s for desktop and mobile respectively. The 95th percentile for these is 4.9s for desktop and 7 seconds for mobile. For completeness, we show the full distribution of both TTFB and time to load in Appendix B.

Finally, we evaluated whether the techniques could be executed concurrently. For instance, instead of a single iframe the attacker can use multiple iframes to load the web pages in parallel. Similarly, asynchronous requests can be launched concurrently, with a limit of 6 concurrent connections per host for HTTP/1.1 [53], and no limit for HTTP/2 or HTTP/3. Based on our experiments on a Macbook Pro with a 2.3GHz i5 processor, we found that on average a 4.27× performance increase could be achieved using 5 concurrent iframes. By further increasing the concurrency, the CPU load would become the bottleneck, resulting in limited additional performance gains and eventually the attack would become noticeable by the user as the laptop's fans would activate. For concurrent fetch operations, a concurrency factor of 20 to 50 could be easily achieved, depending on the latency to the server (higher latency results in longer idle time and thus allows for higher concurrency).

3) Operational cost benefits: In cross-origin vulnerability detection the adversary only needs to host the malicious JavaScript on a web server (∼$30 USD/month). To evaluate the financial benefit for an adversary, we estimate the number of attacks that can be performed given a number of visitors that execute the malicious payload, and calculate the equivalent cost of running the attack on a cloud environment. We consider three types of setups: 1) directly from the cloud provider, 2) using Tor to connect to the target sites, and 3) using a residential VPN. Because the latter two, and especially Tor, introduce additional latency, it takes much more time to launch an attack and thus more servers are required to achieve a similar vulnerability detection rate. For our estimation we take into account the average page visit time (54s [14]), concurrent browser instances on a cloud instance (2 browsers per CPU), latency (based on Tor performance metrics [62] and the CRuX dataset [20]), pricing for cloud services (computing: $0.15USD per hour for 4 vCPUs, egress data: $0.15USD per GB [3]), and various pricing plans for residential VPN services [8]. We report on the option that is most affordable for the adversary given the various parameters.

In Figure 3 we show the estimated cost for cloud and residential VPN services to achieve an equivalent number of detection attempts compared to cross-site vulnerability

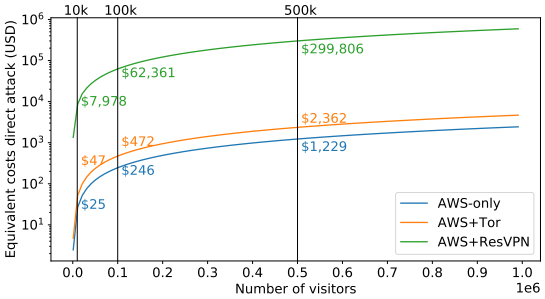

Fig. 3: Estimate of costs that need to be spent to achieve an equivalent number of direct attacks on a cloud instance.

detection performed by a certain number of users (in the graph these range up to 1 million). For reference, Grier et al. find in their 2010 study that spam links on Twitter attracted over 1.6M visitors [21]; Oest et al. find that phishing sites may attract several tens of thousands of visitors within a few days [45]. Note that in the case of the attacks presented in this paper, the website embedding the malicious payload can appear completely legitimate, and thus may be less subject to takedown attempts.

We estimate that the cloud costs that are equivalent to 10,000 page visits is approximately $25 USD when a direct connection is used. However, since the cloud provider's server is directly attacking other websites, it is likely to quickly receive complaints, which would eventually lead to shutting down the attacker's server or even their account. The attacker could instead leverage the Tor network to hide the originating IP address of the attack, preventing targeted websites to correctly attribute the source of the attacks. Due to the additional latency imposed by the Tor network, the attacks would incur a significant performance loss, and the equivalent in cloud costs would almost double: $47 USD for 10k page visits, $472 USD for 100k visits, ... Moreover, as we show in our experiment in Section II-B, traffic from the Tor network is much more likely to be blocked (∼15% sites blocked every request originating from Tor). Hence, we argue that the cross-site vulnerability detection techniques presented in this paper is most closely related to using a residential proxy when launching attacks. Interestingly, the costs for these proxies significantly outweigh the cloud computing costs, and we find that an attacker would have to pay almost $8,000 USD to achieve the equivalent vulnerability detection capacity of 10k page visits. This is in very stark contrast to the cross-site vulnerability detection setup, where the attacker has a fixed cost of ∼$30 USD per month, making it much more economically viable to launch attacks this way.

VI. Defenses

In this section we explore to what extent defenses in the current Web landscape can thwart the threat posed by cross-origin vulnerability detection. Furthermore, we analyze what additional steps can be taken by website administrators and browser vendors to protect websites and their users.

TABLE III: The percentage of cross-origin vulnerability detection attempts that is prohibited by various security features.

		Defenses						
Vulnerability	**Detection technique**	CORB	CORP	COOP	SameSite	CSP	framing protection	**Overall**
XSS	`postMessage()` (iframe)	-	-	-	-	0.17%	24.24%	24.30%
	`postMessage()` (window)	-	-	0.16%	-	0.17%	-	0.34%
	`frames.length` (iframe)	-	-	-	-	-	24.24%	24.24%
	`frames.length` (window)	-	-	0.16%	-	-	-	0.16%
	Prerender	-	-	-	-	0.61%	-	0.61%
SQL injection	Response inflation	72.99%	0.031%	-	-	-	-	73.00%
	Time inflation	-	-	-	-	-	-	-
	Error detection	-	-	-	-	-	24.24%	24.24%
Login brute-force	History API	-	-	0.043%	∼72.99%	-	24.24%	79.54%
SSRF	Web hooks service	-	-	-	-	-	-	-

A. Browser security mechanisms

For each security mechanism we evaluate to what extent it impedes the different cross-origin vulnerability detection techniques. In Table III we show the percentage of attacks that will be blocked by a certain defense; a dash (-) indicates that the defense is ineffective against the technique. For each feature that is enabled through a response header, we will determine for which sites the policy defined in the header will be sufficient to thwart an attack. For instance, a CSP policy with `unsafe-inline` will be ineffective at stopping the XSS detection method using the postMessage API. We base our results on the HTTP Archive dataset [25]. For features that are enabled by default in the browser, we consider the market share of the browsers that support the feature, according to the latest (December 2021) statistics of statcounter [60]: Chrome: 63.8%, Safari: 19.6%, Edge: 3.99%, Firefox: 3.91%, Samsung Internet: 2.85%, Opera: 2.35%. Note that Chrome, Edge, Samsung Internet and Opera are based on the Chromium engine; Safari is based on WebKit and Firefox on Gecko.

CORB. Originally developed as a defense against Spectre attacks [29], Cross Origin Read Blocking (CORB) aims to prevent sensitive resources to be loaded in the renderer process [11]. The CORB mechanism will try to detect the content type of the response, and if it considers it to be potentially sensitive (HTML, XML & JSON), it will stop loading the resource. This defense, which is enabled by default in Chromium-based browsers (72.99% market share), mitigates detection methods that rely on detecting the response size.

CORP. The Cross-Origin Resource Policy (CORP) mechanism has the same goals as CORB but instead requires an explicit opt-in from the developer by setting the appropriate response header [70]. It is supported by all browser engines. We found only 1,794 websites (0.031%) that enabled this header with a `same-site` or `same-origin` value on their homepage.

COOP. To prevent other, cross-origin windows from retaining a reference to the current window, a web developer can leverage the recently introduced Cross-Origin Opener Policy (COOP) [26]. This mechanism will instruct the browser to disassociate two windows by removing any reference they may have to each other. As such, when opening a page with a COOP policy set, it will not be possible to close it, redirect it elsewhere, or access any of its properties. We found 9,101 (0.16%) sites that enabled COOP and thus prevented attacks that relied on opening a new window. For the error-detection and History API techniques we only considered websites that also enabled framing protection: these two techniques can be applied either using an iframe or new window, and COOP alone does not provide sufficient protection for the former. Surprisingly, only 2,490 sites (0.043%) enabled COOP in combination with framing protection.

SameSite cookies. The `SameSite` attribute can be used to indicate whether a cookie should be included in cross-origin requests: when the value is set to `Lax` or `Strict`, cookies will not be included in cross-site requests [39]. On Chromium-based browsers, the default has been set to `Lax`, with other browsers likely to follow soon [10], [56]. In this paper we mainly focus on detecting vulnerabilities in the unauthenticated part of a website, and thus most methods do not rely on cookies as previous studies [55]. One exception is the login brute-force technique leveraging the History API: if the redirect only occurs when the user's cookies are included in subsequent requests, the SameSite=Lax cookies will prevent this detection method. As the HTTP Archive dataset only contains information about unauthenticated page visits, and it is unclear which cookies would be used for authentication, it is infeasible to determine the exact impact. Therefore we estimate the percentage of prevented attacks to be the market share of the browsers that enable the feature by default.

CSP. Through a number of content security policy (CSP) directives, websites can instruct the browser to enforce a variety of aspects [78]. For instance, via the `script-src` directive, website can instruct the allowed source from which scripts can be included. Unless the `unsafe-inline` attribute is present, this is an effective way to stop most XSS vulnerabilities from being exploited. Furthermore, CSP can be used to provide framing protection, which we will consider together with XFO, and to upgrade insecure requests (the most widely used directive) [58]. For the vulnerability detection mechanisms, we find that the techniques based on the postMessage API could be thwarted by CSP, as these will inject JavaScript code and rely on it being executed. Although 175,680 sites (3.02%) enable CSP, we found that only 10,097 (0.17%) enforced a strict policy to thwart reflected XSS vulnerabilities; i.e. no `'unsafe-inline'` nor a wildcard source (e.g. `https://*`). Furthermore, we found that 0.61% of sites would restrict the sources from which images could be included, impeding our prerender-based detection method. Note that the frame-based detection method for XSS is unaffected by any CSP

policy; however, exploiting the vulnerabilities will likely require manual analysis to find a CSP bypass [77].

Framing protection. To prevent a web page to be framed by another, it is possible to use the X-Frame-Options (XFO) header or the `frame-ancestors` directive of CSP. We found that 24.24% of sites enabled framing protection either via XFO or a non-wildcard CSP policy. Although the login brute force detection method leveraging the History API can also be launched with new windows, this may be more noticeable to users and thus an unfavorable option for the adversary. Hence, we consider that framing protection also impedes it.

Overall. Finally, we determine the extent to which certain detection methods are prevented by considering all different defenses. Generally, we find that for each vulnerability there exists at least one cross-origin detection technique that is not stopped in more than 99% of cases. The defenses that have the largest impact are those that are enabled by default: CORB and SameSite cookies, as the adoption of security features by websites is fairly limited. This also provides browsers with the most opportunities to thwart the potential threat of cross-origin vulnerability detection.

B. Limitations

Although we have shown that detecting common vulnerabilities in a cross-origin context is feasible, these still face a number of practical limitations. Most importantly, in comparison to direct attacks, the adversary has less control over the requests that are sent to the target server. For instance, browsers limit the request headers that can be used, or which values it supports, e.g. the `Origin` or `User-Agent` headers cannot be controlled by the adversary. Consequently, vulnerabilities that require the manipulation of these headers, e.g. when the user agent string is logged using a vulnerable Log4j library [79], cannot be detected in a cross-origin context. Similarly, the detection methods can only be applied to applications that use HTTP for communication.

Another limiting factor is that for the detection methods relying on XS-Leaks, the body of the response is not accessible. This means that the adversary cannot iteratively adjust their payload based on the response, and thus a single, uniform payload should be used, which ideally can be applied in most situations. Furthermore, this limitation prevents the adversary from crawling through the site. To overcome this, web rehosting service can be used in conjunction with the XS-Leaks-based methods, where the former is used for website exploration and the latter to detect vulnerabilities.

VII. Related work

Prior work has reported on various malicious operations that are executed in the background when visiting a website. For example, Papadopoulos et al. describe MarioNet, a web-based botnet that leverages various HTML5 features such as service workers to achieve persistent and stealthy infection [48]. This persistence can be used to launch DDoS attacks, store files, or perform cryptocurrency mining. This persistence could also be leveraged to extend the lifetime of our cross-origin vulnerability detection attacks. However, at the time of writing, browsers have limited the background operations of service workers to a maximum of 30 seconds [13], hence we do not consider it a viable option.

In 2006, Lam et al. explored how web browsers could be misused to perform various attacks, such as denial of service and reconnaissance scans [32]. Moreover, they propose worm propagation by launching CSRF attacks that include the malicious payload. In our paper, we show that with more advanced techniques based on XS-Leaks, custom web applications can be scanned for vulnerabilities. Upon compromise the affected websites could be made to serve the malicious payload, a tactic that is known to be successful [52]. Other work on malicious browser operations have focused on forcing unwitting visitors to perform DDoS attacks [31], [49], [47] or cryptojacking [17].

VIII. Conclusion

In this paper we explore the potential threat posed by cross-origin vulnerability detection. We demonstrate that the restrictions imposed by the same-origin policy can be circumvented in three ways: abusing permissive CORS policies, leveraging web rehosting services, and exploiting the various leaks in modern-day browsers. These techniques either provide the adversary with complete access to the response body (CORS & web rehosting services), or relevant side-channel information that reveals whether a vulnerability is present (XS-Leaks). Both have their advantage and disadvantages, and we believe it would likely be most opportune for an adversary to combine both techniques, e.g. using CORS and web rehosting service for content discovery and crawling purposes, and XS-Leaks for the actual vulnerability detection.

These novel ways of detecting common vulnerabilities in websites has a number of advantages over the traditional techniques where an attacker-controlled server is used to directly attack websites. Most prominently is the fact that with cross-site vulnerability detection the IP addresses from which attacks are launched have a high variability and primarily originate from trusted, residential networks. This significantly complicates the detection, e.g. by security services, of such attacks, and may limit the effectiveness of rate-limiting systems. Another critical advantage of these type of attacks is the operating cost for the adversary, which is limited to hosting a single website. Finally, we find that the defense mechanisms that are currently deployed in the web ecosystem, either enabled by default in the browser or opted in to by websites, are mostly ineffective at thwarting these attacks.

Ethical considerations

As a result of our research, we discovered several vulnerabilities, affecting multiple parties. The bugs we uncovered in the browser engines were reported to the relevant vendors. All evaluations of our vulnerability detection techniques were launched against our own servers, hosted on a custom domain. We took necessary precautions to ensure no other users or services were involved in our evaluations.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven, and by the Flemish Research Programme Cybersecurity. This project was partially supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101019206 (TESTABLE).

References

[1] Akamai. API: The attack surface that connects us all. *State of the Internet. Volume 7, Issue 4*, 2021.

[2] Amazon. Amazon EC2 on-demand pricing. https://aws.amazon.com/ec2/pricing/on-demand/, 2022.

[3] Amazon. Amazon EC2 on-demand pricing. https://aws.amazon.com/ec2/pricing/on-demand/, January 2022.

[4] Beeceptor. Rest api mocking and intercepting in seconds. https://beeceptor.com/, January 2022.

[5] Chris Bentzel. Chrome prerendering. https://www.chromium.org/developers/design-documents/prerender, February 2011.

[6] Hugo LJ Bijmans, Tim M Booij, and Christian Doerr. Inadvertently making cyber criminals rich: A comprehensive study of cryptojacking campaigns at internet scale. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1627–1644, 2019.

[7] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *Proceedings of the 16th international conference on World Wide Web*, pages 621–628. ACM, 2007.

[8] bright data. Residential proxies. https://brightdata.com/proxy-types/residential-proxies, 2022.

[9] Nilesh Christopher. Hackers mined a fortune from Indian websites. https://economictimes.indiatimes.com/small-biz/startups/newsbuzz/hackers-mined-a-fortune-from-indian-websites/articleshow/65836088.cms, September 2018.

[10] Chrome Platform Status. Cookies default to SameSite=Lax. https://www.chromestatus.com/feature/5088147346030592, November 2019.

[11] Chromium. Cross-origin read blocking for web developers. https://www.chromium.org/Home/chromium-security/corb-for-developers, 2018.

[12] Chromium. net/base/port_util.cc. https://source.chromium.org/chromium/chromium/src/+/main:net/base/port_util.cc, January 2022.

[13] Chromium. Service worker security FAQ. https://chromium.googlesource.com/chromium/src/+/main/docs/security/service-worker-security-faq.md, January 2022.

[14] Contentsquare. Digital experience benchmark. https://explore.contentsquare.com/2021-benchmark-report, 2021.

[15] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, pages 281–290, 2010.

[16] Kun Du, Hao Yang, Zhou Li, Haixin Duan, and Kehuan Zhang. The ever-changing labyrinth: A large-scale analysis of wildcard DNS powered blackhat SEO. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 245–262, 2016.

[17] Shayan Eskandari, Andreas Leoutsarakos, Troy Mursch, and Jeremy Clark. A first look at browser-based cryptojacking. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 58–66. IEEE, 2018.

[18] Sergiu Gatlan. Google chrome adding malicious drive-by-downloads protection. https://www.bleepingcomputer.com/news/security/google-chrome-adding-malicious-drive-by-downloads-protection/, January 2019.

[19] Nethanel Gelernter and Amir Herzberg. Cross-site search attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1394–1405. ACM, 2015.

[20] Google Developers. Chrome user experience report. https://developers.google.com/web/tools/chrome-user-experience-report, 2021.

[21] Chris Grier, Kurt Thomas, Vern Paxson, and Michael Zhang. @ spam: the underground on 140 characters or less. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 27–37, 2010.

[22] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, volume 1, pages 13–15. IEEE, 2006.

[23] Johannes Henkel. Adding rank magnitude to the CrUX Report in BigQuery. https://developers.google.com/web/updates/2021/03/crux-rank-magnitude, March 2021.

[24] Luan Herrera. Xs-leaks in redirect flows. https://docs.google.com/presentation/d/1rlnxXUYHY9CHgCMckZsCGH4VopLo4DYMvAcOltma0og/, January 2021.

[25] HTTP Archive. The HTTP Archive tracks how the web is built. https://httparchive.org/, January 2022.

[26] Artur Janc, Charlie Reis, and Anne van Kesteren. COOP and COEP explained. https://docs.google.com/document/d/1zDlfvfTJ_9e8Jdc8ehuV4zMEu9ySMCiTGMS9y0GU92k, February 2019.

[27] Chris Kanich, Nicholas Weaver, Damon McCoy, Tristan Halvorson, Christian Kreibich, Kirill Levchenko, Vern Paxson, Geoffrey M Voelker, and Stefan Savage. Show me the money: Characterizing spam-advertised revenue. In *USENIX Security Symposium*, volume 35, 2011.

[28] Lukas Knittel, Christian Mainka, Marcus Niemietz, Dominik Trevor Noß, and Jörg Schwenk. XSinator.com: From a formal model to the automatic evaluation of cross-site leaks in web browsers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1771–1788, 2021.

[29] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[30] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1714–1730, 2018.

[31] Lavakumar Kuppan. Attacking with HTML5. https://media.blackhat.com/bh-ad-10/Kuppan/Blackhat-AD-2010-Kuppan-Attacking-with-HTML5-wp.pdf=, 2010.

[32] VT Lam, Spiros Antonatos, Periklis Akritidis, and Kostas G Anagnostakis. Puppetnets: misusing web browsers as a distributed attack infrastructure. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 221–234. ACM, 2006.

[33] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS 2019, February 2019.

[34] Sangho Lee, Hyungsub Kim, and Jong Kim. Identifying cross-origin resource status using application cache. In *NDSS*, 2015.

[35] Zhou Li, Kehuan Zhang, Yinglian Xie, Fang Yu, and XiaoFeng Wang. Knowing your enemy: understanding and detecting malicious web advertising. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 674–686, 2012.

[36] Xiaojing Liao, Kan Yuan, XiaoFeng Wang, Zhongyu Pei, Hao Yang, Jianjun Chen, Haixin Duan, Kun Du, Eihal Alowaisheq, Sumayah Alrwais, et al. Seeking nonsense, looking for trouble: Efficient promotional-infection detection through semantic inconsistency search. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 707–723. IEEE, 2016.

[37] Long Lu, Roberto Perdisci, and Wenke Lee. Surf: detecting and measuring search poisoning. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 467–476, 2011.

[38] MDN. DOMParser. https://developer.mozilla.org/en-US/docs/Web/API/DOMParser, 2021.

[39] Rowan Merewood. SameSite cookies explained. https://web.dev/samesite-cookies-explained, May 2019.

[40] Xianghang Mi, Xuan Feng, Xiaojing Liao, Baojun Liu, XiaoFeng Wang, Feng Qian, Zhou Li, Sumayah Alrwais, Limin Sun, and Ying Liu. Resident evil: Understanding residential ip proxy as a dark service. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1185–1201. IEEE, 2019.

[41] Seyed Ali Mirheidari, Sajjad Arshad, Kaan Onarlioglu, Bruno Crispo, Engin Kirda, and William Robertson. Cached and confused: Web cache deception in the wild. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 665–682, 2020.

[42] Seyed Ali Mirheidari, Matteo Golinelli, Kaan Onarlioglu, Engin Kirda, and Bruno Crispo. Web cache deception escalates! In *31st USENIX Security Symposium (USENIX Security 22)*, pages 179–196, 2022.

[43] Antonio Nappa, M Zubair Rafique, and Juan Caballero. Driving in the cloud: An analysis of drive-by download operations and abuse reporting.

In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–20. Springer, 2013.

[44] Mark Nottingham. The Cache-Status HTTP response header field. https://httpwg.org/http-extensions/draft-ietf-httpbis-cache-header.html, January 2022.

[45] Adam Oest, Penghui Zhang, Brad Wardman, Eric Nunes, Jakub Burgis, Ali Zand, Kurt Thomas, Adam Doupé, and Gail-Joon Ahn. Sunrise to sunset: Analyzing the end-to-end life cycle and effectiveness of phishing attacks at scale. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 361–377, 2020.

[46] OWASP. OWASP top ten. https://owasp.org/www-project-top-ten/, 2021.

[47] Yao Pan, Jules White, and Yu Sun. Assessing the threat of web worker distributed attacks. In *2016 IEEE Conference on Communications and Network Security (CNS)*, pages 306–314. IEEE, 2016.

[48] Panagiotis Papadopoulos, Panagiotis Ilia, Michalis Polychronakis, Evangelos P Markatos, Sotiris Ioannidis, and Giorgos Vasiliadis. Master of web puppets: Abusing web browsers for persistent and stealthy computation. *arXiv preprint arXiv:1810.00464*, 2018.

[49] Giancarlo Pellegrino, Christian Rossow, Fabrice J Ryba, Thomas C Schmidt, and Matthias Wählisch. Cashing out the great cannon? on browser-based ddos attacks and economics. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[50] Pipedream. A modern request bin to inspect any event. https://requestbin.com/, January 2022.

[51] PortSwigger. Cross-site scripting (XSS) cheat sheet: Using srcdoc with entities. https://portswigger.net/web-security/cross-site-scripting/cheat-sheet#using-srcdoc-with-entities, 2022.

[52] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, Nagendra Modadugu, et al. The ghost in the browser: Analysis of web-based malware. *HotBots*, 7:4–4, 2007.

[53] Push Technology. Browser connection limitations. https://docs.pushtechnology.com/cloud/latest/manual/html/designguide/solution/support/connection_limitations.html, 2022.

[54] Konrad Rieck, Tammo Krueger, and Andreas Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 31–39, 2010.

[55] Iskander Sanchez-Rola, Davide Balzarotti, and Igor Santos. Baking-Timer: Privacy Analysis of Server-Side Request Processing Time. In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[56] Iskander Sanchez-Rola, Davide Balzarotti, and Igor Santos. Cookies from the Past: Timing Server-Side Request Processing Code for History Sniffing. *ACM Dgital Threats: Research and Practice Journal (DTRAP)*, 2020.

[57] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies. In *USENIX Security Symposium (USENIX Security)*, 2017.

[58] Saptak Sengupta, Tom Van Goethem, and Nurullah Demir. Web almanac: Security chapter. https://almanac.httparchive.org/en/2021/security, December 2021.

[59] John Snow. WhatsApp and facebook ticket giveaways: Viral fraud. https://www.kaspersky.com/blog/whatsapp-fake-tickets-scam/25419/, 2019.

[60] statcounter. Browser market share worldwide - december 2021. https://gs.statcounter.com/browser-market-share, December 2021.

[61] Avinash Sudhodanan, Soheil Khodayari, and Juan Caballero. Cross-origin state inference (COSI) attacks: Leaking web site states through xs-leaks. *arXiv preprint arXiv:1908.02204*, 2019.

[62] The Tor Project. Tor metrics: Performance. https://metrics.torproject.org/onionperf-latencies.html, 2022.

[63] Kurt Thomas, Chris Grier, Dawn Song, and Vern Paxson. Suspended accounts in retrospect: an analysis of twitter spam. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 243–258, 2011.

[64] Phani Vadrevu and Roberto Perdisci. What you see is not what you get: Discovering and tracking social engineering attack campaigns. In *Proceedings of the Internet Measurement Conference*, pages 308–321, 2019.

[65] Tom Van Goethem, Gertjan Franken, Iskander Sanchez-Rola, David Dworken, and Wouter Joosen. Sok: Exploring current and future research directions on xs-leaks through an extended formal model. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, pages 784–798, 2022.

[66] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1382–1393. ACM, 2015.

[67] Tom Van Goethem, Najmeh Miramirkhani, Wouter Joosen, and Nick Nikiforakis. Purchased fame: Exploring the ecosystem of private blog networks. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 366–378, 2019.

[68] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. Timeless timing attacks: Exploiting concurrency to leak secrets over remote connections. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1985–2002, 2020.

[69] Anne van Kesteren. Cross-origin resource sharing. http://www.w3.org/TR/cors, 2010.

[70] Anne van Kesteren. Fetch: Living standard. https://fetch.spec.whatwg.org/, August 2022.

[71] Said Varlioglu, Bilal Gonen, Murat Ozer, and Mehmet Bastug. Is cryptojacking dead after coinhive shutdown? In *International Conference on Information and Computer Technologies*, pages 385–389. IEEE, 2020.

[72] Verizon. Data breach investigations report. https://www.verizon.com/business/resources/reports/dbir/, 2021.

[73] W3Techs. Usage of content management systems. https://w3techs.com/technologies/overview/content_management/all, January 2022.

[74] David Y Wang, Stefan Savage, and Geoffrey M Voelker. Juice: A longitudinal study of an seo botnet. In *NDSS*, 2013.

[75] Takuya Watanabe, Eitaro Shioji, Mitsuaki Akiyama, and Tatsuya Mori. Melting pot of origins: Compromising the intermediary web services that rehost websites. In *NDSS*, 2020.

[76] Webhook. Webhook.site. https://webhook.site/, January 2022.

[77] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. CSP is dead, long live CSP! on the insecurity of whitelists and the future of content security policy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1376–1387. ACM, 2016.

[78] Mike West. Content security policy level 3. https://www.w3.org/TR/CSP3/, June 2021.

[79] Free Wortley, Chris Thompson, and Forrest Allison. Log4shell: RCE 0-day exploit found in log4j 2, a popular java logging package. https://www.lunasec.io/docs/blog/log4j-zero-day/, December 2021.

[80] XS-Leaks Wiki Contributors. Xs-leaks wiki: Navigations. https://xsleaks.dev/docs/attacks/navigations/, October 2020.

[81] XS-Leaks Wiki Contributors. Xs-leaks wiki. https://xsleaks.dev/, 2022.

APPENDIX

A. Detailed description subresource detection XS-Leak

In Figure 4 we show a detailed timeline of the operations that happen with a web page with or without subresource is loaded in an iframe. In the attack, the adversary will create a new iframe and first navigate it to `about:blank`. At this point, the iframe inherits the origin of the embedding window, namely `https://attacker.com`, and polling for DOM properties is still possible. Next, the attacker will navigate the iframe to the target page, on a different origin (`https://target.com`). This will cause the renderer process to fetch the response body of this page. Once the HTML has been downloaded, the renderer will make the iframe navigate to the origin of the target page. When this happens, it is no longer possible to access DOM properties, as this would violate the same-origin policy. Instead, the browser will throw an exception, which can be caught to know when the change of origin occurred. As soon as this happens, the attacker will set the `src` attribute back to `about:blank`. However, at this time to renderer is still parsing the HTML of the target page. We find that if this parsing process encounters no subresources that need to be loaded, a `load` event will be fired on the iframe before it is navigated to `about:blank`. Otherwise, the loading of the subresources would delay the `load` event, causing it to never be fired.

B. CrUX timing distributions

In Figure 5, we show the distribution of the time-to-first-byte for both desktop and mobile, based on the data available in the Chrome User Experience Report. The time-to-first-byte indicates the time between the start of the initialization of a connection and the first byte of the actual response being received. As such, it includes the multiple hops required to establish a TCP connection and TLS session. Note that this is likely an overestimation for the time to fetch a response as the multiple hops required to establish a connection are only required once; for the following requests the connection can be reused.

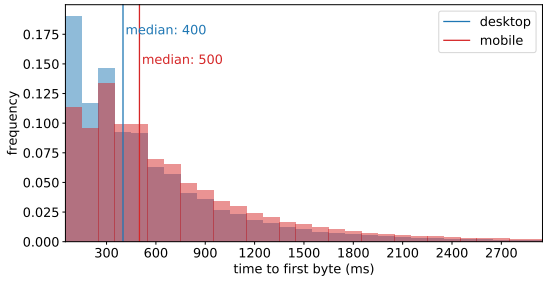

Fig. 5: Distribution of the time to first byte.

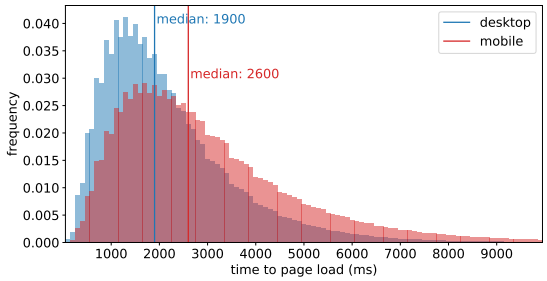

Fig. 6: Distribution of the time to the `load` event.

Similarly, in Figure 6 we show the distribution of how long users take to load a web page, measured based on the time that the `load` event is fired. This coincides with the time that all subresources have been loaded by the browser.

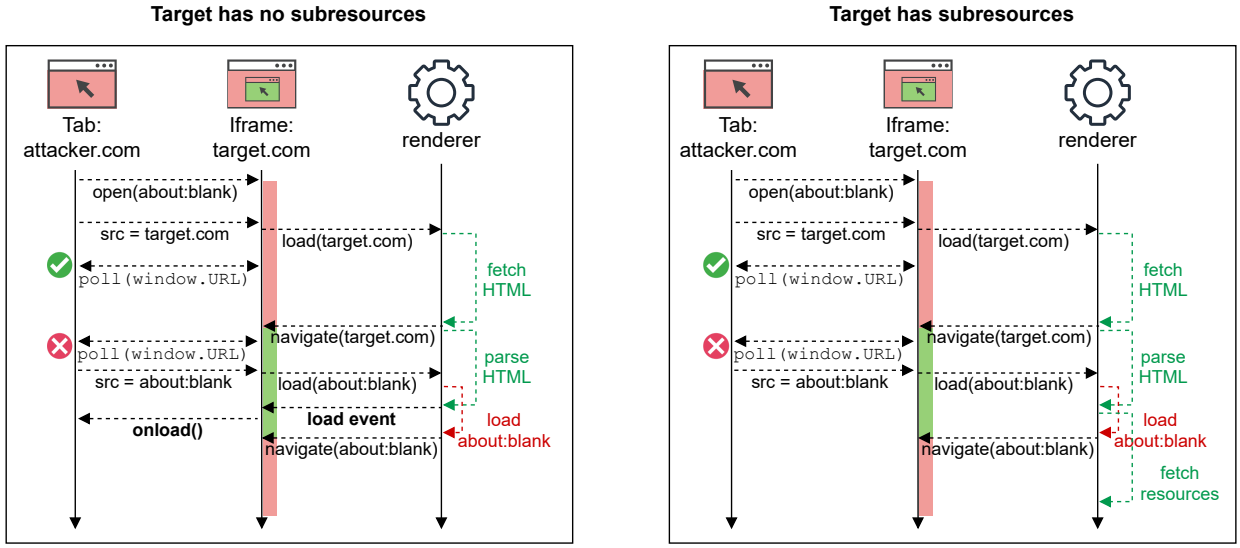

Fig. 4: Timelines of 2 iframe loads for a document without subresources (left) and one with (right). The bar in the center indicates the current origin of the iframe (attacker.com is red and target.com is indicates as green).