Mind the CORS

Matteo Golinelli
University of Trento
Trento, Italy
matteo.golinelli@unitn.it

Elham Arshad
University of Trento
Trento, Italy
elham.arshad@unitn.it

Dmytro Kashchuk
University of Trento
Trento, Italy
dmytro.kashchuk@studenti.unitn.it

Bruno Crispo
University of Trento
Trento, Italy
bruno.crispo@unitn.it

***Abstract*—Cross-Origin Resource Sharing (CORS) is a mechanism to relax the security rules imposed by the Same-Origin Policy (SOP), which can be too restrictive for websites that rely on cross-site data exchange for their functioning. CORS allows trusting origins different from the website domain despite the presence of a strict SOP using a series of HTTP headers. This mechanism is supported by all modern browsers and is extensively adopted by websites. In CORS, servers are responsible for validating the value of the Origin header and deciding whether or not to trust it. For this reason, developers must be thorough in coding this process not to introduce security issues. We carried out a large-scale analysis on the Tranco Top 50k to measure the prevalence of various implementation flaws due to errors or simplifications in Origin validation and found that of the 6,862 websites using CORS, 2,014 (29.4%) have at least one flaw. We next exploit the vulnerabilities introduced by these CORS flaws in a realistic real-world scenario from the point of view of two attacker models with varying capability levels, evaluating the conditions necessary for a successful attack and its consequences. We show how these flaws enable attackers to perform Denial of Service and steal victims' sensitive data and security tokens that can then be used to mount subsequent attacks. We conclude that CORS is an effective but complicated mechanism and its use should be carefully evaluated by website operators not to risk introducing severe security issues in their systems.**

***Index Terms*—CORS, SOP, cross-site requests, cross-origin, cache poisoning.**

I. Introduction

The Same-Origin Policy (SOP) is a web security mechanism that allows restricting access to resources on a site from origins other than the site itself. The origin of a site is defined by the three values of protocol, host and port. However, if a website relies on interchanging data with third-party websites with different origins, the SOP may be too restrictive and break its functionality. For websites that wish to maintain cross-site information exchange with certain third-party websites without relinquishing the use of the SOP as a protection mechanism, the Cross-Origin Resource Sharing (CORS) mechanism was introduced [1]. CORS is based on two HTTP headers in response to cross-site requests: "Access-Control-Allow-Origin" (ACAO), which allows indicating whether to trust the origin included in the request, and "Access-Control-Allow-Credentials" (ACAC), which allows the server to instruct on whether authentication cookies and any authorization headers may be attached to requests by the browser [2].

The enforcement of the rules established by CORS is delegated to the client browser, while the server is responsible for verifying the value of the origin of requests and the subsequent decision on whether or not to trust it. For this reason, the logic that verifies the value of the origin is crucial for the security of the website. Since the origin verification in CORS is programmed by the application developers, there is a high possibility of introducing flaws that lead to trusting websites that can potentially be controlled by malicious actors, compromising the website's security. The simplest case of dangerous CORS configuration is when the value of the request origin is simply copied into the ACAO header of the response, effectively trusting every possible origin. Other dangerous configurations may be introduced by errors in the creation of regular expressions, by the use of prefixes or suffixes in the checks, or by allowing the value null.

Previous research has focused on measuring the prevalence of CORS flaws in the wild, without investigating the actual exploitability of such flaws in a realistic real-world scenario [3]–[5].

The goal of this research is to answer the following questions.

(Q1) What are the required conditions for CORS flaws to be exploitable in a real-world scenario by existing attacker profiles in standard threat models?

(Q2) What are the consequences of the exploitation of such vulnerabilities for the victims?

To answer these questions, we first conduct a large-scale analysis of the homepages and login pages of the websites in the Tranco Top 50k to measure the prevalence of several variations of CORS implementation flaws. Since CORS is not configured the same for all resources on a site but can potentially vary for each URL, we decided to test two pages from each site. Of 6,862 websites using CORS, we found 2,014 (29.4%) that have at least one CORS flaw. We then partly automate and replicate the attacks enabled by such flaws in a real-world scenario from the perspective of two types of attacker models, distinguished by the different levels of capabilities they possess: *1) web attacker*: has the least power (and therefore it is the most dangerous); can operate a website with a generic domain for which they possess a valid HTTPS certificate. *2) related-domain attacker*: controls a website hosted on a subdomain of the target website.

We discover that the consequences of the exploitation of CORS flaws can enable attackers to steal personal and potentially sensitive information of authenticated users, along with stealing security tokens of both authenticated and non-authenticated visitors, which can later be used to carry out sub-

sequent attacks, such as CSRF (*Cross-Site Request Forgery*) and login CSRF. Moreover, we investigate the possibilities for attackers to exploit CORS flaws to achieve Denial of Service.

To summarize, we make the following contributions:

- We conduct a large-scale analysis to measure the prevalence of CORS flaws in the Tranco Top 50k ranking. We find that 29.4% of the websites that employ CORS have at least one CORS flaw.
- We identify the conditions necessary for CORS flaws to be exploitable by two attacker models with different power levels against victims using modern up-to-date browsers with default security settings. We find that the default security settings of some browsers enable the exploitability of CORS flaws.
- We develop a methodology to semi-automatically replicate the attacks enabled by CORS flaws in a realistic real-world scenario and analyze their consequences. We show that exploiting the CORS flaws enables attackers to steal victims' personal and security information and achieve DoS.

Availability. The source code of our tools is available on the author's website[1].

II. Background & Related Works

In this section, we present an overview of how Cross-Origin Resource Sharing (CORS) works and discuss related concepts such as Same-Origin Policy (SOP), cross-site requests, and state-changing requests, and describe the risks involved in cross-origin communications.

A. Access Control Policy in Web

The Same-Origin Policy (SOP) is a fundamental concept in web application security, on which the main principle of web security is based. The SOP sets access restrictions on web resources (including sensitive information), isolating them and providing boundaries from other websites with different origins. This rule does not apply to websites with the same origin: two websites are considered from the same origin if and only if all the following three values are exactly the same: protocol, host, and port [1].

As the way SOP builds a protective wall for websites, it became too restrictive for a large portion of websites that need to communicate beyond the boundary of their current origin, for example, their subdomains. Due to the complexity of the web environment, more websites are relying on other websites for exchanging data to provide better functionality to users. To address this problem, in 2006, the W3C introduced a mechanism, called Cross-Origin Resource Sharing (CORS), to relax the SOP allowing cross-origin requests and sharing of resources between websites with different origins [3]. CORS, an extension of the XMLHttpRequest API, functions through a set of HTTP headers enforced by the client to provide the permissions to access the selected resources in cross-origin requests. The server performs the validation, authorization, and access restrictions, while it is the browser's responsibility to support these HTTP headers to enforce the restrictions. This protocol has been adopted by all major browsers (e.g., Chrome, Firefox, IE), and has been widely adopted by websites.

We take websites A and B as an example to describe CORS in a real scenario. Website A requests a resource from website B and the SOP will not grant this access unless website B is CORS configured and responds with an "Access-Control-Allow-Origin" (ACAO) header, indicating website A as a trusted party to access that resource. To check whether the server is configured to use CORS, the browser sends a preflight request, i.e., an OPTIONS HTTP request that includes the three headers "Access-Control-Request-Method", to discover which methods are supported, "Access-Control-Request-Headers", and "Origin" [6]. ACAO header supports a single origin within a CORS response and if a website wants to allow any origin, the header sets to * wildcard. An additional header called "Access-Control-Allow-Credentials" (ACAC) is available which, if set to true, allows credentialed requests (i.e., with authentication cookies and authorization headers attached). Importantly from a security point of view, credentialed requests with a * wildcard are forbidden [7].

B. Cookies

Cookies are a mechanism to introduce the concept of "state" into the HTTP protocol, which would otherwise be stateless. Web servers use cookies to maintain the visitors' sessions; therefore, their confidentiality and integrity are crucial to keeping the session secure. Compromising the confidentiality or integrity of a user's session can enable the stealing of sensitive data, hijacking the account, and replacing the session (e.g., login CSRF [8]). Web servers use the HTTP Set-Cookie header to save cookies on the visitor's user agent, characterised by a name, value and possible attributes; the browser will automatically attach the cookies to subsequent requests [9].

1) SameSite: Cross-site attacks are successful when the browser includes valid cookies in the requests performed from the malicious site; therefore, an effective solution to cross-site attacks is to set restrictions on the cookies' scope. The SameSite attribute in the Set-Cookie HTTP response header introduces three policies, None, Lax, and Strict, that instruct the browsers on which requests to attach cookies [10].

None This policy specifies that cookies are sent in all outgoing requests, including first-party and cross-site ones. This policy corresponds to the default policy before the introduction of the SameSite attribute. When setting SameSite=None, the Secure cookie attribute must also be set, otherwise, cookies will be blocked. The Secure attribute specifies that a cookie must only be attached to HTTPS requests.

Lax The Lax policy tries to increase the usability of the website while maintaining its security. Lax cookies are sent for requests issued by top-level navigation (e.g., clicking on a link), but not for sub-requests (e.g., requests to load media files). This is the default value for cookies when the SameSite attribute is not specified in the Set-Cookie header on

[1] https://github.com/Golim/mind-the-cors

Chromium-based browsers (including Chrome, Edge, Opera, and Brave), while Firefox and Safari default the attribute to None [2]. Note that some Firefox tracking protections may isolate third-party cookies for cross-site requests, preventing cookies from being sent even if they do not specify the SameSite attribute [11].

Strict This value is more stringent than other values for attaching cookies to outgoing requests. It prevents the browser from sending the cookies in all cross-site browsing contexts, even those with safe methods, and only allows requests from the same website to include cookies.

C. Cross-Site Attacks

A large part of web security is involved in the study of cross-site (XS) attacks, in which a victim visits a malicious site that makes authenticated cross-site HTTP requests from the victim's browser to a vulnerable website. In fact, depending on the SameSite attribute of cookies, explained in detail in section II-B, the browser automatically attaches cookies to the requests to the sites from which they were created. Cross-site attacks allow to steal victims' personal information, take control of their accounts or perform actions on their behalf on vulnerable web applications. The SameSite attribute, if set correctly, prevents this kind of attacks [10].

D. Cache poisoning

A web cache is a component of the web architecture that stores copies of web resources such as HTML pages, images, and other media files. When a user requests a resource, the web cache first checks if it has a valid copy. If the cache contains a copy of the requested resource, it returns it to the user instead of requesting it from the origin server, reducing the load on the origin server and improving the performance. In addition to storing the response body, the cache generally saves the response headers. The cache key is a unique identifier used to store and retrieve cached resources. The cache key is typically based on the URL of the requested resource, but it may also include other factors, such as the query string parameters and cookies. If the cache key does not include the value of the Origin header in the request, an attacker can perform a Denial of Service (DoS) attack exploiting CORS flaws by sending a request with a malformed Origin header that is mistakenly trusted by the web cache. The cache then stores the response, including the ACAO header with the malformed Origin value, and returns it to subsequent visitors that request the same resource. When the browser receives the response, it compares the Origin header of the request with the ACAO header of the response, and since the two values do not match, the browser blocks it, resulting in a DoS.

E. Related Works

CORS is a relatively new security mechanism, and several academic and non-academic researchers have identified various security problems. Some studies have highlighted common flaws of CORS ([4], [5], [12]). Kettle [12] provides a summary of several CORS flaws identified throughout his penetration testing experiences. Müller [4] takes the different CORS flaws and measures their prevalence on the Alexa top 1M websites, while Evan J performs a measurement of the arbitrary origin reflection in [5], showing a high number of vulnerable websites in the Alexa top 1M. Chen et al. [3] provide an empirical study for CORS security, finding some new issues in the design and implementation of CORS: they craft the size and value of the requests headers and body, leading to Remote Code Execution (RCE), file upload CSRF and attacks on binary protocol services, and endangering the privacy of the user. They also analyze the risky relationship between websites through CORS including third-party and subdomain websites. In addition, they measure the CORS flaws of 50k websites and the frameworks they use [3]. Meiser et al. in [7] construct a graph of interconnected trust relationships between websites considering existing cross-communication methods, namely postMessage, CORS and domain relaxation. They focus specifically on the dangers that the interconnected network of trust could cause and investigate the attack surface. They estimate the damage of XSS exploitation that usually occurs when websites trust each other on the interconnected web.

Previous work on the same topic focused only on identifying CORS flaws, while in this work we investigate what the attackers can do by exploiting CORS flaws. We study and replicate the possible attacks to steal personal information and cause Denial of Service.

III. Threat Model

Due to the complexity of the web, security professionals must consider all angles and choose suitable attacker models to cover possible web attacks. Therefore, as a fundamental assumption of this research, we take two types of attackers from the web security literature: the web attacker and the related-domain attacker. Our threat model, unlike previous research on CORS flaws, does not depend on the presence of XSS vulnerabilities on other websites and does not assess the security of third-party websites that could affect the target website. We describe the types of attackers with respect to our attacks based on CORS implementation flaws.

Web Attacker. The most well-known attacker model in the web security literature is the web attacker, which is of great concern to security professionals. A web attacker operates at least one website that responds to any HTTP(S) requests with malicious content and mounts attacks through standard HTML and JavaScript code. This attacker has no privileged access or control over the network. A web attacker could be anyone who registers a domain and, possibly, obtains an HTTPS certificate [13]. Web browsers are designed to protect users even when they visit a malicious website. Therefore, we assume that the web browser correctly implements the web standards and has default settings according to its version.

Related-domain Attacker. This attacker is a more powerful web attacker that controls a malicious website hosted on a sibling domain of the target website. A sibling domain is

[2]We tested all browsers using https://samesitetest.com/.

a domain that shares a suffix long enough with one of the target websites that is not present in a public database of suffixes, such as facebook.com or bbc.co.uk. For example, if we take “example.com” as the target website, we assume that a related domain attacker has control over “evil.example.com”. This attacker is more powerful compared to a simple web attacker because cookies are generally shared between a domain and its siblings [13]. Attacks on related domains might seem uncommon in the real world, as it is assumed that the owner of “example.com” would never allow control over “evil.example.com” to untrusted parties. However, recent research has shown that the takeover of subdomains is a serious and widespread security risk [14].

IV. CORS Flaws

Frequently, web developers generate dynamic CORS policies deployed in web applications that dynamically validate the value of the request’s “Origin” header. If these policies are not implemented correctly on the server side, the web application might unintentionally trust domains that it is not intended to trust and hinder the SOP enforcement. Due to the complexity of this validation and the pitfalls in CORS design, different types of flaws can arise in CORS implementations. Following we list the types of CORS flaws that we considered in our research. The flaws are compiled based on previous research and supplemented with new flaws we found through our experiments [3].

Arbitrary origin reflection: The basic way to configure CORS while dynamically generating the policies is to blindly reflect the “Origin” header value of the request in the “Access-Control-Allow-Origin” header in the responses. This configuration trusts any domain that performs the request.

Prefix matching: the server ignores the ending characters and trusts any domain prefixed with a trusted domain. For instance, a server wants to trust “victim.com” and allows “victim.com.attacker.com”.

Suffix matching: the server only checks if the ending characters match the trusted domain, or their own domain, as a way to allow all the subdomains. For example, if a server wants to allow “victim.com”, it mistakenly trusts any domains that end with “victim.com” as well, e.g., “attackervictim.com”.

Not escaping ‘.’: when the validation is performed using a regular expression, the developer might forget to escape “.” characters in the configuration. For instance, “victim.com” wants to allow “www.victim.com”, but allows “wwwAvictim.com” as well.

“null” value: the “Origin” header was first proposed as a mitigation against CSRF attack, and CORS reuses this header [15]. One of the essential conditions for CORS security is that the “Origin” header value cannot be forged in a cross-origin request, but this is not always true in reality. RFC 6454 states that a request from a privacy-sensitive context should set the “Origin” header to “null”, even though it does not provide an explicit definition for privacy-sensitive context [15]. Moreover, CORS standards do not define the “null” value clearly. In reality, browsers send the ’null’ value from multiple sources, like iframe sandbox scripts. To share data with these types of sources, a developer must allow the null value in their configuration, setting “Access-Control-Allow-Origin: null” and “Access-Control-Allow-Credentials: true”. By doing so, an attacker can easily forge the “Origin” header by sending a cross-origin request from an iframe sandbox in the browser. Consequently, websites with this flaw can be read by any other websites.

HTTPS site trusts HTTP domain: some CORS configurations do not take the protocol (scheme) into account and cause HTTPS sites to trust HTTP ones.

Arbitrary subdomains: most applications decide to allow access from all their subdomains (even non-existent ones). Additionally, websites might allow access from various third-party websites, including all their subdomains.

End matching: this differs from suffix matching in that the website is not only checking for the trusted domain but also for the scheme. Consequently, a website that wants to trust “https://victim.com” mistakenly trusts “https://attacker.com/https://victim.com”. This flaw cannot be exploited by a web attacker because the browser will never generate such a value for the Origin header; however, this variation can be exploited to achieve DoS by manually crafting an HTTP request with a malformed Origin.

V. Methodology

We present our measurement methodology in three phases: 1) Collection, 2) Detection, and 3) Exploitation, as shown in Fig. 1.

A. Collection

The first phase aims at finding the web pages to test. Our goal is to identify two URLs for each site: the landing page and the login page. In fact, different paths might have different CORS configurations. We chose the homepage and the login page to have data on both a resource that can be visited with authentication (by logging into the sites) and one that should only be accessible by unauthenticated visitors. This also allowed us to assess the impact of CORS flaws against unauthenticated victims.

We developed a tool that uses Python and a Selenium web browser to identify the websites using CORS. Our tool checks both the home pages and the login pages. Specifically, we visited the domains using both the HTTPS and HTTP protocols in both forms of the URL, with or without “www.” prepended. To identify the login pages, the tool uses a specially designed heuristic that relies on keyword matching both on the URL and the HTML code of the web page.

Next, we test all collected web pages to check whether they use CORS or not. For each web page to test, we try to force the use of CORS by including the “Origin” header with a genuine value in an HTTP request and checking the presence of the ACAO header in the response. If the ACAO header is present, the web page is using CORS and can be tested for possible flaws.

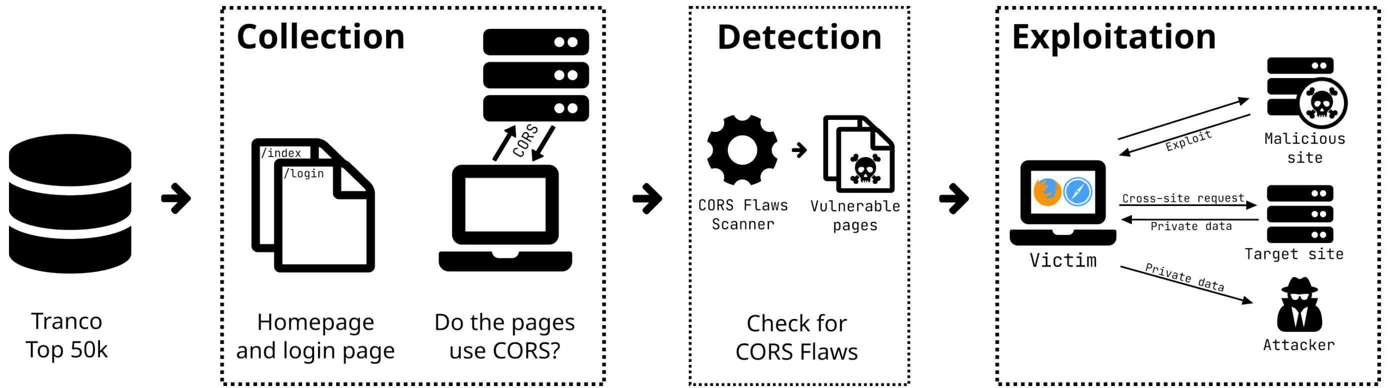

Fig. 1. High-level visualization of our methodology in three phases: collection of candidate pages to test, detection of CORS flaws, and flaws exploitation.

B. Detection

In this phase, we test all extracted web pages from the previous phase for the CORS flaws listed in Section IV. We developed a tool called *CORS Flaws Scanner* that sends one HTTP request for each flaw to each web page, where the value of the Origin header is mutated accordingly to the variation. If the mutated Origin value is reflected in the response ACAO header, the web page is misconfigured to the tested variation. All the HTTP requests to test for CORS flaws are performed using the *python requests* library, providing the User-Agent of a legitimate Chrome web browser to simulate a genuine user visiting the web page using a browser.

C. Exploitation

As discussed in Section IV, flaws in the validation of the Origin header value for CORS might lead to the emergence of security vulnerabilities. In this section, we describe different cross-site attacks that we replicated against websites with CORS flaws from the perspective of the two attacker models described in Section III, against victims using modern up-to-date browsers (i.e., Firefox, Chromium-based browsers, and Safari).

Web Attacker To carry out the attack, the web attacker simply creates a website with a domain name that is trusted by the CORS (mis)configuration of the target website and generates an HTML page containing JavaScript code that performs a cross-site request to the target website. As we will discuss in more detail later, websites that are vulnerable to all the variations we test, excluding "HTTPS site trusts HTTP domain", "arbitrary subdomains" and "end matching", allow a web attacker to register a domain that will be mistakenly trusted by the target website.

We partly automated the exploitation of these vulnerabilities by creating a tool that works as follows:

1) The tool binds the attacker's domain name (mistakenly trusted by the target site) to *localhost* using the */etc/resolv.conf* file, a configuration file used by the DNS resolver of several operating systems, simulating an attacker with controlling the domain. In a real-world scenario, an attacker would register the domain using a registrar.
2) The tool generates an exploit HTML containing the JavaScript code presented in Listing 1. This code performs an XHR (*XMLHttpRequest*) request to the flawed web page on the target website instructing the browser to include credentials (i.e., cookies, authorization headers) and stores the response content. The code prints the response content in the console (line 5 in Listing 1), while in an attack scenario, the data would be exfiltrated to an attacker's controlled server and saved in a database.
3) The exploit HTML code is served by a web server running in localhost that can be accessed using the previously bound domain name and, if necessary, providing an encrypted connection with HTTPS using TLS certificates specifically created and installed in the browser (in a real-world attack scenario, an attacker could generate a certificate using free services such as Let's Encrypt [16]).
4) Finally, the automation uses a puppeteer-controlled browser to open the login page of the target website, where the tester manually logs into the test victim account, and then is redirected to the exploit web page using the previously bound domain. Due to CORS flaws, the response to the XHR request, which contains the sensitive data of the logged-in victim, will be exposed to the attacker. In a real-world scenario, the victim would already be authenticated on the target website and the response content would be exfiltrated to the attacker.

Listing 1. A sample of an automatically generated JavaScript code that performs the cross-site request

```
1 <script>
2    var url = 'URL_PLACEHOLDER';
3    var req = new XMLHttpRequest();
4    req.addEventListener('load', () =>
5       console.log(req.responseText));
6    req.open('get', url, true);
7    req.withCredentials = true;
8    req.send();
9 </script>
```

The only task that must be performed manually by the tester is logging into the website since automating this action is extremely complicated due to every site implementing it differently.

To exploit the *null_value* flaw we use a modified JavaScript code that creates an exploit web page where the same script presented in Listing 1 is included in an `iframe`. In fact, the Origin header of HTTP requests performed from inside an iframe is set to the value “null”.

This attack is mitigated by websites by correctly setting the SameSite attribute of authentication cookies, therefore, for the attack to work against a flawed target, the following conditions must be met:

1) An attacker must be able to register a domain name mistakenly trusted by the CORS configuration of the target website on a registrar.
2) The CORS configuration of the target website must allow credentialed requests (i.e., “ACAC: true”).
3) The website must not set the *SameSite* attribute of authentication cookies, or it must be set to the *None* value.

Moreover, the victim must use a web browser that does not implement the *Lax-by-default* policy for the SameSite attribute when not otherwise specified (e.g., Firefox, Safari).

To test this attack on each potentially exploitable website (i.e., the websites that meet the aforementioned conditions), it is necessary to simulate an authenticated victim, which is why it is required to conduct the registration and login procedure on each website to be tested to create a dummy account populated with bogus information. Since the number of websites to be tested is high and this procedure is time-consuming, we decided to test only those websites that allow registration and login using Google and Facebook OAuth. To identify the websites that support OAuth with at least one of these two IdPs (*Identity Providers*), we developed a tool that, given the URL of the login page (previously identified in the Collection phase described in Section V-A), identifies any OAuth button present in the page. Specifically, this tool is based on a Selenium browser and Python’s *BeautifulSoup* [17] library to crawl through all HTML tags of type *a*, *input* and *button* and checks whether they contain some specific keywords (e.g., “Log in with”, “Continue with”). The tool also checks whether any URL in the HTML code of the web page is an OAuth URL of the two providers, using a regex system.

Related-domain Attacker This attacker can perform the same attack, but instead of controlling any arbitrary domain, the malicious actor must have control of a sub-domain of the target website. To simulate the exploitation of these vulnerabilities, we used the same automation as for the web attacker, mimicking an attacker with the control of a sub-domain by using the */etc/resolv.conf* file.

D. Unauthenticated Victim

Motivated by the findings of Mirheidari et al., who in [18] show how even web pages publicly accessible to unauthorized visitors can contain valuable secrets for an attacker to bypass security mechanisms or mount subsequent attacks, we statically analyzed the content of all vulnerable login pages to detect the presence of common types of sensitive information. To detect the sensitive information, we used regular expressions on the dynamic parts of login pages, i.e., those parts of HTML code that change when the web page is requested multiple times using different browsers. The dynamic parts of login pages can be for instance CSRF tokens, OAuth state parameters and CSP nonce. As shown in [8], stealing the state parameter allows attackers to mount login CSRF attacks.

TABLE I
EXPERIMENT STATISTICS: NUMBER OF SITES AND PAGES THAT WE VISITED THAT USE CORS AND ARE VULNERABLE TO AT LEAST ONE CORS FLAW. PERCENTAGES FOR EACH ROW ARE CALCULATED OVER THE NUMBER OF VISITED SITES OR PAGES IN THE *Visited* COLUMN.

	Visited	Use CORS	Vulnerable
Websites	39,067	6,862 (17.6%)	2,014 (5.2%)
Total web pages	58,815	7,681 (13.1%)	2,350 (4.0%)
Homepages	39,064	5,328 (13.6%)	1,536 (3.9%)
Login pages	19,751	2,353 (11.9%)	814 (4.1%)

E. Cache Poisoning

Finally, we tested the possibility of exploiting CORS flaws to cause Denial of Service. As explained in Section II, for this attack to be possible, the website must present at least one CORS flaw and the cache must not include the value of the Origin header in the cache key. To check for sites that present these characteristics and are therefore vulnerable to DoS we tested all sites that present CORS flaws with a specific heuristic algorithm. We first send a request with a malformed Origin that is mistakenly trusted by the website (i.e., its value is reflected in the ACAO response header). Next, we send a second request, this time with a genuine Origin of the website, and check whether the value of the response matches the genuine Origin or the malformed one. If the ACAO value of the second response includes the modified Origin value, the website is vulnerable to cache poisoning and prone to DoS. To avoid poisoning actual resources that could be accessed by genuine users of the tested websites, we used specific cache-busting techniques, such as adding random query parameters. This also allows us to avoid the accessed resources from being already cached when we send the first request.

VI. EXPERIMENT

In this section, we present the results of the empirical analysis and discuss them in detail. We conducted a large-scale experiment and performed different attacks using CORS flaws in the wild through modern browsers with default settings from the perspective of the two types of attacker models.

A. Results

We conducted our experiment over the websites included in the Tranco Top 50k [19] generated on 04 April 2023 [3].

[3]Available at https://tranco-list.eu/list/W9J39.

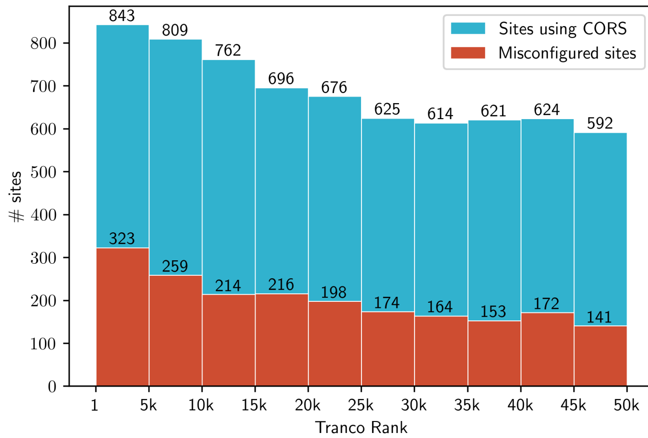

Fig. 2. Distribution of websites using CORS and misconfigured to at least one variation with respect to their Tranco ranking in 5k bins.

As described in Section V-A, we first crawled each website to identify its homepage and login pages. In total, we identified 39,064 homepages and 19,751 login pages on a total of 39,067 websites. For 10,933 websites, we could not identify the homepage or login page due to various problems. Primarily, we encountered errors such as request time-outs, hosts not resolvable by the DNS, and unreachable destinations. Additionally, some domains lacked an actual homepage (e.g., domains that serve static resources and answer HTTP requests only for specific paths). In total, we successfully tested 58,815 pages, 7,681 of which (on 6,862 different websites) responded with CORS headers.

Using our *CORS Flaws Scanner* tool, we tested all the collected web pages to detect possible CORS flaws. As shown in Table I, we found that 2,014 sites are misconfigured to at least one variation of the ones listed in Section IV, for a total of 2,350 web pages. Table II shows the number of sites misconfigured to specific variations. Note that in this table, we only count the more generic variation and not the most specific ones. For example, if a website is misconfigured to "arbitrary origin reflection", we only count it for that, even though all the other variations would test positive.

Fig. 2 illustrates the distribution of websites using CORS that are misconfigured to at least one variation in relation to their Tranco ranking. The chart suggests that the adoption of CORS is slightly related to the popularity of the website, while the number of websites with CORS flaws is only related to the number of websites using CORS.

B. Exploitation

In this section, we describe the attacks replicated in a real-world scenario, carried out from the point of view of the two types of attackers described in Section III.

Due to the large number of impacted websites, it was not possible for us to perform the attacks on all websites. For this reason, we replicated the attacks on only a subset of the sites affected by CORS flaws. It is important to note that, as we will explain in the section on the web attacker, for this attacker the exploitation of CORS flaws is not always successful and requires additional conditions to be verified for each target website. For this reason, we tested a statistically relevant number of sites and present in Table IV the percentage of sites where we were able to carry out the attack successfully.

By contrast, the exploits carried out by the related-domain attacker do not require any additional conditions for the attack to be successful.

Web Attacker This attack is performed by the web attacker who controls a website for which they possess an HTTPS certificate and on which they host malicious code; the attacker uses social engineering techniques to induce the unsuspecting victim to visit the malicious website. As described in Section V-C, three conditions are required for this attack to be possible. The variations that meet the first condition are lines 3 to 7 in Table II. Moreover, if the target issues cookies without setting the SameSite attribute, the victim must use a browser that does not implement the *Lax-by-default* policy. Table III shows the number of pages and websites with CORS flaws that meet the conditions, with a total of 157 websites.

However, it is not enough for these conditions to be met for an attacker to be able to mount a successful attack. In fact, these conditions are measured when the visitor is not authenticated, while the attack is carried out against an authenticated victim. For this reason, we selected 30 websites that implement Facebook or Google OAuth to log in, registered a dummy account as the victim, and performed the attack as described in Section V-C. We successfully performed the attacks on 21 websites, while for the other 9 websites, the attack failed for different reasons (e.g., requests blocked by the WAF, enforcing different CORS policies when the visitor is authenticated). We were able to steal the victim's sensitive data on several websites, categorized in the "Authenticated" column in Table IV.

Related-domain Attacker This attacker model can exploit the "Arbitrary subdomain" variation (line 2 in Table II). For this attack to be successful, no further conditions are necessary, and all 848 websites are vulnerable to cross-site attacks. We successfully replicated this attack on a selection of sites by simulating an attacker in control of a subdomain using our methodology described in Section V-C.

C. Unauthenticated Victim

The column *Unauthenticated* in Table IV presents the number of security tokens detected in the source code of vulnerable login pages, accessed by a non-authenticated visitor using a clean browser. By stealing CSRF tokens or the OAuth state assigned to a victim, an attacker is able to perform login CSRF attacks, forcing the user into logging into an attacker's account on the targeted website.

D. Cache Poisoning

As described in Section II, if the Origin header is not included in the cache key of web caches, the website might be susceptible to Denial of Service. Using the methodology that

TABLE II
NUMBER OF PAGES AND SITES MISCONFIGURED TO THE TESTED VARIATIONS. PERCENTAGES ARE CALCULATED OVER THE TOTAL NUMBER OF FLAWED PAGES OR SITES, PRESENTED IN THE LAST ROW OF THE TABLE FOR EACH COLUMN.

ID	Variation	Homepages	Login pages	Sites
1	**HTTPS trusts HTTP**	1391 (90.6%)	682 (83.8%)	1786 (88.7%)
2	**Arbitrary subdomain**	606 (39.5%)	396 (48.6%)	848 (42.1%)
3	**Arbitrary origin reflection**	484 (31.5%)	248 (30.5%)	655 (32.5%)
4	**“null” value**	476 (31.0%)	240 (29.5%)	640 (31.8%)
5	**Prefix matching**	110 (7.2%)	65 (8.0%)	159 (7.9%)
6	**Non escaped dot**	123 (8.0%)	45 (5.5%)	146 (7.2%)
7	**Suffix matching**	85 (5.5%)	24 (2.9%)	91 (4.5%)
8	**End matching**	307 (20.0%)	177 (21.7%)	422 (21.0%)
	Total	**1536**	**814**	**2014**

TABLE III
NUMBER OF PAGES AND SITES THAT MEET THE CONDITIONS REQUIRED FOR THE ATTACK TO BE SUCCESSFUL FOR THE WEB ATTACKER LISTED IN SECTION V-C. EACH ROW IS A SUBSET OF THE PRECEDING ROW. THE VARIATIONS REFERENCED IN THE FIRST CONDITIONS ARE PRESENTED IN TABLE II.

	Homepages	Login pages	Sites
Total misconfigured	**1536**	**814**	**2014**
1) Misconfigured to variations 3 to 7	718 (46.7%)	379 (46.6%)	959 (47.6%)
2) Allow credentials	345 (22.5%)	258 (31.7%)	524 (26.0%)
3) No SameSite attribute	77 (5.0%)	117 (14.4%)	157 (7.8%)
All conditions	**73** (4.8%)	**112** (13.8%)	**157** (7.8%)

TABLE IV
NUMBER OF VULNERABLE HOMEPAGES AND LOGIN PAGES THAT LEAK SENSITIVE INFORMATION OF AUTHENTICATED AND UNAUTHENTICATED VICTIMS RESPECTIVELY. NOTE THAT *personal information* CANNOT BE FOUND ON UNAUTHENTICATED PAGES.

Total tested	Authenticated (30 homepages)	Unauthenticated (814 login pages)
Personal Information	15 (50.0%)	—
CSRF Token	5 (16.7%)	163 (20.0%)
CSP Nonce	0 (0.0%)	63 (7.7%)
OAuth State	1 (3.3%)	22 (2.7%)

we developed and described in Section V-C, we identified 45 sites that are vulnerable to this attack. An attacker can abuse any variation that we tested for because it is enough for the injected value to differ from the Origin value of genuine HTTP requests.

VII. DISCUSSION & CONCLUSION

We answer our first research question (Q1), showing how theoretical flaws introduced by CORS flaws can be exploited in practice to break the security of websites. We analysed the conditions necessary for two different attacker models with varying levels of capabilities to exploit the security issues introduced by CORS flaws and the consequences of these attacks, mainly related to the confidentiality of victims' data. Moreover, we analysed how stealing security tokens assigned to victims may allow attackers to mount subsequent attacks against them, also impacting the integrity and availability. The web attacker, with the lowest level of prior capabilities required, can only exploit the flaws when three conditions are met and against victims using browsers without the *Lax-by-default* policy, but the severity of the resulting attacks is greater than that of the others. This attack is only made possible by the fact that some modern browsers, such as Firefox and Safari, with an estimated combined usage of 16% [20], do not implement the *Lax-by-default* policy proposed by Google precisely to mitigate this type of flaw [21]. Firefox initially supported and then discontinued this policy from version 69 on, in favour of backwards compatibility [22]; however, as our research shows, it also sacrificed part of its users' security. Firefox users can enable the Lax-by-default policy from the settings to protect themselves against web attackers; however, this still does not protect against the other two attackers, for which no simple solution is available. Moreover, the Lax-by-default policy will most likely be adopted by all browsers in the near future: Firefox has already started to do so in some nightly and beta versions [23], [24], while for Safari it is not yet clear [25].

Related-domain attackers can exploit one type of flaw, provided they control at least one subdomain of the target website, with severe consequences for the confidentiality of victims' data on the main domain.

We also answer (Q2) by showing that the exploitation of these vulnerabilities leads to stealing victims' sensitive data. Moreover, we analysed the consequences of such an attack against unauthenticated victims, stealing the security tokens assigned to them on vulnerable login pages. Finally, we investigate how CORS flaws can lead to Denial of Service, finding that 45 sites in our dataset fail to include the Origin value in the cache key and are therefore affected.

29.4% of the websites using CORS had at least one flaw.

Of these sites, in particular, the vast majority of websites trust the HTTP version of their domain; nearly half allow arbitrary subdomains and one in three trusts the value null or reflects the value of the request origin in the response. These results suggest that developers are either unaware of the potential security consequences of these dangerous behaviours or underestimate them. We believe that devolving the burden of validating the Origin programmatically on the server, instead of introducing a policy-based system enforced directly by the browser, has introduced an inherent complexity to the CORS mechanism, which is thus prone to human error with severe consequences for web security.

Ethical considerations During the execution of the attacks that we reproduced and described, we always used specifically created test accounts as victims. No users of any website were impacted by our attacks. We employed cache-busting techniques to avoid poisoning resources potentially accessed by genuine users of the target websites. During all experiments, we minimised the impact on targeted websites by limiting the number of requests performed to the minimum possible; moreover, we performed no disruptive attacks. Although we have never disclosed the domains of sites impacted by these vulnerabilities to any third party and in this article, we will proceed with the responsible disclosure to all websites that provide security contact in the coming weeks.

Acknowledgements

This work has been partially supported by the EU Horizon project DUCA (GA 101086308) and CrossCon (GA 101070537). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or CINEA. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] J. Schwenk, M. Niemietz, and C. Mainka, “Same-Origin policy: Evaluation in modern browsers,” in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 713–727. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schwenk

[2] J. Wilander, “CORS-safelisted request headers should be restricted according to RFC 7231 · Issue #382 · whatwg/fetch.” [Online]. Available: https://github.com/whatwg/fetch/issues/382

[3] J. Chen, J. Jiang, H. Duan, T. Wan, S. Chen, V. Paxson, and M. Yang, “We still Don’t have secure Cross-Domain requests: an empirical study of CORS,” in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1079–1093. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/chen-jianjun

[4] J. Müller, “On Web-Security and -Insecurity: CORS misconfigurations on a large scale,” Jul. 2017. [Online]. Available: https://web-in-security.blogspot.com/2017/07/cors-misconfigurations-on-large-scale.html

[5] E. J, “Misconfigured cors,” 2016. [Online]. Available: https://ejj.io/misconfigured-cors

[6] WHATWG, “Fetch Standard.” [Online]. Available: https://fetch.spec.whatwg.org/

[7] G. Meiser, P. Laperdrix, and B. Stock, “Careful Who You Trust: Studying the Pitfalls of Cross-Origin Communication,” in *ASIACCS 2021 - 16th ACM Asia Conference on Computer and Communications Security*, ser. 16th ACM Asia Conference on Computer and Communications Security, Hong Kong / Virtual, China, Jun. 2021. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03021256

[8] E. Arshad, M. Benolli, and B. Crispo, “Practical attacks on login csrf in oauth,” *Computers & Security*, vol. 121, p. 102859, 2022.

[9] A. Barth, “HTTP State Management Mechanism,” RFC 6265, Apr. 2011. [Online]. Available: https://www.rfc-editor.org/info/rfc6265

[10] S. Khodayari and G. Pellegrino, “The state of the samesite: Studying the usage, effectiveness, and adequacy of samesite cookies,” in *43rd IEEE Symposium on Security and Privacy (S&P ’22)*, 2022. [Online]. Available: https://publications.cispa.saarland/3504/

[11] “Enhanced tracking protection in firefox for desktop.” [Online]. Available: https://support.mozilla.org/en-US/kb/enhanced-tracking-protection-firefox-desktop

[12] J. Kettle, “Exploiting cors misconfigurations for bitcoins and bounties,” 2016. [Online]. Available: https://portswigger.net/research/exploiting-cors-misconfigurations-for-bitcoins-and-bounties

[13] A. Bortz, A. Barth, and A. Czeskis, “Origin cookies: Session integrity for web applications,” *ACM Transactions on Internet Technology (TOIT)*, vol. 2, 05 2012.

[14] M. Squarcina, M. Tempesta, L. Veronese, S. Calzavara, and M. Maffei, “Can i take your subdomain? exploring Same-Site attacks in the modern web,” in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2917–2934. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/squarcina

[15] A. Barth, “The Web Origin Concept,” RFC 6454, Dec. 2011. [Online]. Available: https://www.rfc-editor.org/info/rfc6454

[16] “Let’s Encrypt.” [Online]. Available: https://letsencrypt.org/

[17] “Beautiful Soup Documentation — Beautiful Soup 4.9.0 documentation.” [Online]. Available: https://www.crummy.com/software/BeautifulSoup/bs4/doc/

[18] S. A. Mirheidari, M. Golinelli, K. Onarlioglu, E. Kirda, and B. Crispo, “Web Cache Deception Escalates!” in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/mirheidari

[19] V. L. Pochat, T. V. Goethem, S. Tajalizadehkhoob, M. Korczynski, and W. Joosen, “Tranco: A research-oriented top sites ranking hardened against manipulation,” in *Proceedings 2019 Network and Distributed System Security Symposium*. Internet Society, 2019. [Online]. Available: https://doi.org/10.14722/ndss.2019.23386

[20] “Global Desktop Browser Market Share for 2022.” [Online]. Available: https://kinsta.com/browser-market-share/

[21] M. West, “Incrementally Better Cookies,” Internet Engineering Task Force, Internet-Draft draft-west-cookie-incrementalism-01, Mar. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-west-cookie-incrementalism/01/

[22] “SameSite cookies - HTTP | MDN.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite

[23] “Changes to SameSite Cookie Behavior – A Call to Action for Web Developers – Mozilla Hacks - the Web developer blog.” [Online]. Available: https://hacks.mozilla.org/2020/08/changes-to-samesite-cookie-behavior

[24] “[meta] Enable sameSite=lax by default.” [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1617609

[25] “Cookies default to SameSite=Lax - Chrome Platform Status.” [Online]. Available: https://chromestatus.com/feature/5088147346030592