UNIVERSITÀ DEGLI STUDI
DI TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE
ICT International Doctoral School

Analysis of Oauth and CORS vulnerabilities in the wild.

Elham Arshad

Advisor

Prof. Bruno Crispo

Università degli Studi di Trento

July 2022

Abstract

Thanks to the wide range of features offered by the World Wide Web (WWW), many web applications have been published and developed through different libraries and programming languages. Adapting to new changes, the Web quickly evolved into a complex ecosystem, introducing many security problems to its users. To solve these problems, instead of re-designing the Web, the vendors added the security patches (protocols, mechanisms)to the Web platform to provide a more convenient and more secure environment for web users. However, not only did these patches not completely resolve the security problems, but their implementations also introduced other security risks unbeknownst to website operators and users.

In this thesis, I propose a novel research on two different security patches to understand and analyze their deployment in real-world scenarios and discover the unseen, neglected factors and the elements involved in exploiting their use: one security protocol, OAuth, and one security mechanism, CORS. As this thesis is based on offensive approaches, I develop automated methodologies, including novel strategies for analyzing and measuring the security qualities of the OAuth protocol and CORS mechanism in real-world scenarios.

Keywords

Protocol, mechanism, OAuth, privacy, CSRF, CORS, Cross-Origin Resource Sharing, Same-Origin Policy, SOP.

Acknowledgments

I would like to thank my advisor, Bruno Crispo, for his support and valuable insights during my Ph.D. career. I would like to thank my reviewers, Alessio Merlo and Aaron Visaggio Corrado, for reviewing my Ph.D. dissertation and their comments.

I am thankful for working with the brilliant people , Matteo Golinelli, Giuliano Turri, Michele Benolli.

Special thanks to my brothers, Saleh and Kazem, for supporting me during this time. And also I thank my whole family, specially my mom and my sisters, for their support and patience during all these years.

Last but not least, I thank my best friend, Dr. Farzaneh Ghalamzan, who stayed by my side all this time and helped me to successfully finish my Ph.D.

Contents

1 Introduction **1**

1.1 Motivation 2

1.1.1 An example of a protocol (OAuth) 2

1.1.2 An example of a mechanism (CORS) 4

1.2 Thesis Contribution 5

1.3 Thesis Structure 6

2 Background **7**

2.1 Authorization in the web 8

2.1.1 OAuth protocol 9

2.2 Referrer Policy 23

2.2.1 Cross-Domain Referrer Leakage (CDRL) 23

2.3 Access Control Policy in Web 25

2.4 Cross-Site Attacks 31

2.4.1 CSRF 31

2.5 Cookies 36

2.6 Web Cache 37

3 Related Works **41**

3.1 OAuth 42

3.1.1 Formal Approaches 42

3.1.2 Empirical studies 43

3.2 CORS 50

4 OAuth **53**

4.1 Motivation 54

4.1.1 CSRF 55

4.2 Threat Models 57

4.2.1 Enabling factors 58

4.3 Methodology 60

4.3.1 Phase 1: Target Selection 60

4.3.2 Phase 2: Measurement Setup 61

4.3.3 Phase 3: OCSRF Discovery 62

4.3.4 Ethical Consideration 65

4.4 Analysis 67

4.4.1 Measurement Overview 67

4.4.2 Results 71

4.4.3 Discussion 81

4.5 Conclusion 95

5 CORS **97**

5.1 Motivation 98

5.2 Threat Model 99

5.3 Methodology 101

5.3.1 Phase1: Collection 101

5.3.2 Phase2: Detection 101

5.3.3 Phase3: Exploitation 102

5.4 Analysis 107

5.4.1 Measurement Overview 108

5.4.2 Results 109

5.4.3 Discussion 112

5.5 Cache poisoning through CORS 115

5.6 Conclusion 116

6 Conclusion **119**

Bibliography **120**

List of Tables

4.1 Login scenarios handled by the crawler 69

4.2 Number of exploitable sites in Facebook by OCSRF for each attack scenario 72

4.3 Number of exploitable sites in Google by OCSRF for each attack scenario 73

4.4 Classification of exploitable sites in Facebook by OCSRF - The first category of candidates (with Absence of `state` parameter) 74

4.5 Classification of exploitable sites in Google by OCSRF - The first category of candidates (with Absence of `state` parameter) 74

4.6 Classification of exploitable sites in Facebook by OCSRF - The second category of candidates (with Presence of `state` parameter) 75

4.7 Classification of exploitable sites in Google by OCSRF - The second category of candidates (with Presence of `state` parameter) 77

4.8 Number of exploitable sites with custom header for each attack scenario 79

4.9 Classification of exploitable sites in Google by OCSRF - Constant `state` parameter 88

4.10 Classification of exploitable sites in Facebook by OCSRF - Constant `state` parameter 88

5.1 Browsers that default SameSite attribute to Lax when not specified [12]. *Chromium-based include: Chrome, Chromium, Edge, Opera 105
5.2 Experiment statistics. 110
5.3 Number of sites misconfigured to the tested variations. Percentages are calculated over the number of misconfigured sites (1823). 110
5.4 Number of sites that meet the conditions required for the attack to be successful for the web attacker listed in Section 5.3.3. The variations referenced in the first conditions are presented in Table 5.3. 111
5.5 Number of vulnerable sites and login pages that leak sensitive information of authenticated and unauthenticated victims respectively. 111

List of Figures

2.1 The OAuth 2.0 protocol flow 11
2.2 RFC 6749 - Representation of the authorization code flow . . 15
2.3 RFC 6749 - Representation of the implicit flow 20

4.1 The main steps involved in the CSRF attack against the redirect-uri 57
4.2 Abstract view of OCSRF detection methodology. 60
4.3 Abstract view of OCSRF detection methodology. 64
4.4 Classification of login `state` parameter length 90

5.1 High-level visualization of our methodology in three phases: a collection of candidate pages to test, detection of CORS flaws, and flaws exploitation. 102
5.2 Distribution of websites using CORS and misconfigured to at least one variation with respect to their Tranco ranking in 5k bins. 108
5.3 Distribution of websites using CORS and misconfigured to at least one variation with respect to their Tranco ranking in 5k bins. 109
5.4 CORS error after hitting the cache [6] 115

Chapter 1

Introduction

Since its introduction in 1989, the use of the World Wide Web (WWW) has been expanding to cover a greater demand, e.g., B2B (business-to-business), B2C (business-to-client),information-sharing, healthcare, and government services. With the internet becoming available to the public, more people began sharing sensitive information online.

Malicious entities exploit the web ecosystem as a potential source of revenue in order to steal sensitive data from people and governments. Already by 2003 as the Symantec analysis of network-based attacks [76] reported, eight out of the top ten attacks were associated with Web applications, stating that port 80 was the most frequently attacked TCP port. The report suggested that the vulnerabilities in web applications are the easiest to exploit. Since then, the reliability and security of Web applications have become an increasingly important concern due to the dynamic nature of the Web.

These issues are caused by several factors, one of which is web ecosystem complexity. At first, the Web was initially designed to deliver data to others, but over the years, it quickly evolved into a complex platform due to meeting rising demands. On top of such a complex platform, more sophisticated applications have developed, and as a result, web specifications have also changed rapidly, along with browsers and web-development languages, to

win market share. Unfortunately, the first design of the Web has not adapted enough to this rapid change in order to secure its users, and instead of re-designing this huge ecosystem from scratch, it has been patched over the years by introducing revised versions of existing protocols, adding new ones and/or issuing new mechanisms.

This way of patching problems may guarantee the parties' security on paper (at the level of the specifications), but their real-world implementations created a whole different situation. Since these patches have been utilized by a vast number of software vendors with a different training backgrounds, this situation has brought other types of security issues into the web ecosystem. The vendors are obliged to adapt to those patches (e.g., protocols and mechanisms) to enforce the security of their users, and even though there are many guidelines widely used by developers, for their correct implementation, web applications are still vulnerable due to 1) many unseen, involving human factors and elements, and 2) the guidelines not being implemented properly.

1.1 Motivation

There are many proposed patches (protocols, mechanisms) to guarantee the confidentiality and integrity of users interacting with web applications developed in this vast, complex ecosystem. For this thesis, I studied and analyzed: 1) the most commonly used security protocol for authorization (OAuth) as an example of a security protocol, and 2) the almost new and less studied mechanism for cross-access policy (Cross-Origin Resource Sharing [for short, CORS]) as an example of a mechanism.

1.1.1 An example of a protocol (OAuth)

One of the nowadays pervasively used protocols is OAuth 2.0, an industry-standard protocol for authorization adopted by many websites worldwide to

support both user authentication and authorization for third-party applications.

OAuth was released in 2012 as RFC 6749 and is more streamlined, less complicated, and more accessible for developers to implement, and supports a wide range of third-party applications, such as websites and applications running on a browser, mobile, desktop, or appliance devices[24].

With the boom of software-as-a-service and social networking, the OAuth protocol is being deployed by more and more commercial websites to protect many web resources; Over a billion users employ their Facebook or Google accounts to sign into more than one million websites.

The protocol has been widely studied, and its theoretical and practical security has been covered extensively by the literature. OAuth was designed to enhance several aspects of the former client-server authorization model.

Given the popularity of the OAuth protocol and the proliferation of third-party applications, the risk of compromised implementations of OAuth can be critical, and it becomes essential to understand how secure the OAuth-based system genuinely is. The recent studies on some well-known OAuth vulnerabilities show that many application or authorization servers are still vulnerable, even though many third-party application developers implemented recommended and proposed security countermeasures by standard – with the knowledge that OAuth protocol is very complicated for average developers to comprehend[39].

Most previous works either are based on manual analysis to discover new weaknesses in OAuth implementations or suggest tools to recognize a set of specific and known attacks in OAuth deployments. Therefore, motivated by the prevalence of OAuth vulnerabilities in the wild, the security analysis and testing of OAuth-based systems adaptively and systematically is needed to evaluate its large-scale implementation to measure the impact of the severe vulnerabilities in the wild.

1.1.2 An example of a mechanism (CORS)

One of the common web security mechanisms is the Same-Origin Policy (SOP) which 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 [96].

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 [119].

The enforcement of the rules established by CORS is delegated to the client browser, while the server is responsible of 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 application developers program the origin verification in CORS, there is a high possibility of introducing flaws that lead to trusted 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 copied into the ACAO header of the response, effectively trusting every possible origin. Other dangerous config-

urations may be introduced by errors in the creation of regular expressions, by using prefixes or suffixes in the checks, or by allowing the value null.

1.2 Thesis Contribution

It is critical to define the extent of security vulnerabilities associated with these current patches on the Web due to the growing reliance of users on the Web.

In this thesis, I investigated the feasibility and effectiveness of novel approaches to measure and reduce the security risks introduced by current patches for website vendors and their users.

Regarding the OAuth protocol, we presented the most comprehensive set of test cases to exploit OCSRF vulnerabilities, including novel attack strategies that stress all possible client-side status. They complement and integrate the guidelines provided by documents such as [112, 74] in helping OAuth developers to mitigate implementation mistakes.

We designed a repeatable methodology and conducted automated analysis on 395 high-ranked sites of two representative IdP: Facebook and Google, to assess the prevalence of CSRF attacks against the `redirect_uri` in OAuth implementations.

The analysis discovered that about 36% of targeted sites are still vulnerable and detected about 20% more well-hidden vulnerable sites utilizing the novel attack strategies. We analyzed other CSRF mitigation for all the implemented attack scenarios and tested the impact of the attacks on mitigation, showing how inconsistent they are in different situations.

Regarding CORS, we discovered 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 the subsequent attacks (such as CSRF and login CSRF).

We conducted a large-scale analysis to measure the prevalence of CORS flaws in the Tranco Top 50k ranking. We find that 30.2% of the websites have at least one CORS flaw.

We developed a methodology to exploit candidate websites with CORS flaws and replicate the attacks in a realistic real-world scenario. We then analyzed the consequences of exploiting CORS flaws and discussed potential solutions to mitigate them.

We identified additional conditions necessary for CORS flaws to be exploited by attackers against victims using modern browsers with default security settings.

1.3 Thesis Structure

The remainder of this thesis is organized as follows. Chapter 2 presents the required background. Chapter 3 provides the literature review of the previous research on the OAuth protocol and CORS mechanism. Chapter 4 presents the research on the OAuth protocol, including the design and implementation of the repeatable methodology and the novel attack scenario in the wild. We propose our methodology for large-scale analysis of CORS flaws in Chapter 5. Finally, Chapter 6 concludes the thesis.

Chapter 2

Background

In this chapter, we provide useful information relating to this study on the security issues in OAuth protocol and Cross-origin resource sharing (CORS), containing some background on OAuth protocol, OpenID Connect, Referer policy, Access control policy on the web, Cross-site attacks, and Cookies values.

2.1 Authorization in the web

Authentication is used when a server needs to be certain who has the right to access their website or information, and the user or computer has to prove its identity to the server or client, usually entailing the use of a username and password. Authentication does not determine which actions an individual can perform or which files an individual can access. Authentication only establishes the legitimacy of the system or individual.

Authorization is the process by which a server determines whether or not a client is authorized to use a resource or access a file and ensures that users have appropriate privileges to access the requested resources.

For this matter, some authorization and authentication protocols have been introduced. However, before authorization protocols, a common way for providing access to an account to a third-party application was to simply give it the password and let it act as the user.

This pattern of applications obtaining user passwords obviously has a number of problems. Since the application would need to log in to the service as the user, these applications would often store users' passwords in plain text, making them a target for harvesting passwords[124].

Once the application has the user's password, it has complete access to the user's account, including having access to capabilities such as changing the user's password. Another problem was that after giving an application the password, the only way the user would be able to revoke that access was by changing her password, something that users are typically reluctant to do. [85]

From the implementation side, if the user changed their service credentials, every single one of the connected applications then broke because they could no longer log in.

A final problem is the all-or-nothing of permissions. An application either

had complete access to a user's account and could perform all of the actions that the user could, or none because they did not give the application their credentials.[50]

Naturally, many services quickly realized the problems and limitations of this model and sought to solve this quickly by proposing a standard for API access control to be used by any system to address these concerns, such as OAuth, SAML[36], and OpenID Connect[95].

As an authentication framework, the user does not have to handle her own authentication systems, and as an authorization framework, it enables third-party applications to grant access to resources from the services. [45]

Authorization is typically combined with authentication so that the server knows who is requesting access. In most web applications, developers combined OpenId Connect as authentication and Oauth as authorization for securing the resources on the web.

2.1.1 OAuth protocol

OAuth [57] is a delegation protocol that is useful for transferring authorization decisions across a network of web-enabled applications and APIs, which enables a third-party application to gain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service or by allowing the third-party application to obtain access on its own behalf.[109].

The authorization is made without sharing the user's passwords and lets the user repeal an application's access to their account. It actually works by delegating user authentication to the service which hosts the user account and authorizing third-party applications to access the user account. OAuth protocol provides authorization flows for web and desktop applications and mobile devices. In OAuth protocol, there are four roles, which are explained in detail as follows [59]:

- **Resource Owner: User**: The user who allows an application to access their account is called the resource owner. The application only has access to the area that assumes by the owner (e.g., read or write access).

- **Authorization Server**: The server issues access tokens to the client after successfully authenticating the resource owner and obtaining authorization. It verifies the identity of the user and then issues access tokens to the application.

- **Resource Server**: The server who is responsible for protected resources in case of having the valid access token of the client starts to answer the request. From an application developer perspective, the service's API is able to do the task of resource and authorization server. Both of these jobs are composed and recourse to them as a Service or API role.

- **Client (Application)**: The client is the application that requires access to the user's account. This action takes place when a user is authorized, and the API must validate the authorization.

The abstract OAuth 2.0 flow illustrated in Figure 2.1 describes the interaction between the four roles and includes the following steps:

1. The client requests authorization from the resource owner. The authorization request can be made directly to the resource owner (as shown) or preferably indirectly via the authorization server as an intermediary.

2. The client receives an authorization grant, which is a credential representing the resource owner's authorization, expressed using one of four grant types defined in this specification or using an extension grant type. The authorization grant type depends on the method used by the client to request authorization and the types supported by the authorization server.

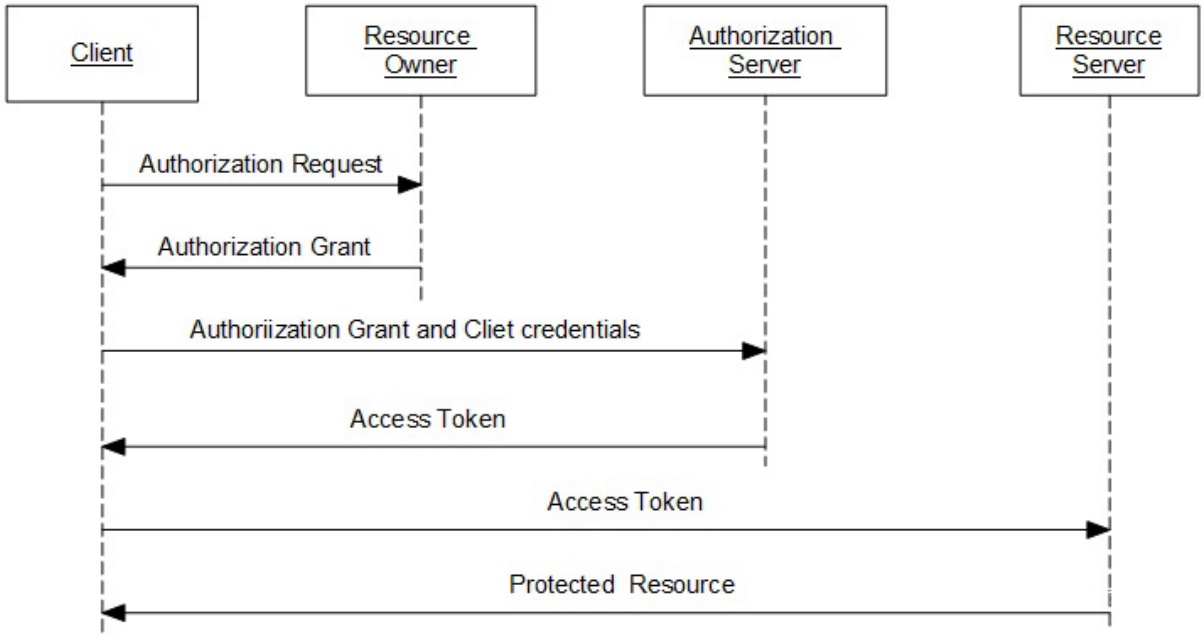

Figure 2.1: The OAuth 2.0 protocol flow

3. The client requests an access token by authenticating with the authorization server and presenting the authorization grant.

4. The authorization server authenticates the client and validates the authorization grant, and if valid, issues an access token.

5. The client requests the protected resource from the resource server and authenticates it by presenting the access token.

6. The resource server validates the access token and, if valid, serves the request.

Application Registration The client must register with the API before it can use OAuth 2.0, during which the API gathers security-critical information about the client, including the client's redirect URI (redirect URI), i.e., the URI to which the user-agent is redirected after the API has generated the authorization response and sent it to the client via the user-agent.

During registration, the API issues the client with a unique identifier

(client id) and, optionally, a secret (client secret). If defined, the client secret is used by the API to authenticate the client in the Authorization Code Grant flow.[59]

Authorization Grant An authorization grant is a credential representing the resource owner's authorization (to access its protected resources) used by the client to obtain an access token; The OAuth 2.0 specification defines four different grant types – authorization code, implicit, resource owner password credentials, and client credentials [35].

- **Authorization Code Mode.** The authorization server initially gives an authorization code for starting the flow between the client and the resource owner. In this case, the client resource owner will not receive authorization directly. This means that a link will be produced between the resource owner and authorization server and the result of this action is that the resource owner will have an authorization code to continue communication with the client.

  Exactly before establishing a link between the resource owner and the client for the authorization code and after authorization of the resource owner, the authorization will be given to the resource owner.

  As mentioned before, the client will not obtain the resource owner credential, which is the pass-through authorization server and resource owner. The authorization code contains important security interests, such as client authentication and access token transfer directly from the resource owner's user-agent to the resource owner.

- **Implicit Grant Mode.** The implicit grant is the simple authorization code flow customized for clients implemented in a browser which are using a scripting language like JavaScript. In the implicit flow, an access token is given to the client directly. In this situation, the authorization code will not issue anymore.

The authorization server does not authorize the client until the access token publishes to the client. In some cases, the client identity can be inquired by redirection URI that is used to deliver the access token to the client.

It is possible that the access token is displayed to the resource owner or other application with an approach to the resource owner's user-agent. Implicit grants increase the performance and responsiveness of some clients (such as a client accomplished as an in-browser application) because this action declines the number of circular trips necessary to take an access token. However, this convenience should be weighed against the security implications of using implicit grants, especially when the authorization code grant type is available.

- **Resource Owner Password Credentials Mode.** In this method, the user presents her/his credential for an API straight to a client. After that, the client is able to authenticate to API on the user's behalf and regain an access token. This style is predesignated for highly-trusted clients, including the operating system of the user's device or highly-privileged applications, or if the prior two methods are not feasible to perform (e.g., for applications without a web browser).

- **Client Credentials Mode.** In opposition to the style shown above, this method works without the user's interplay. In lieu, it is begun by a client instead of giving and bringing an access token to access the resources of the client at an API. Facebook lets clients to usage the client credential mode to gain access to reports of their advertisements efficiently. It is generally used for programs that do not have any direction to communicate with programs for machine-to-machine (like background services and daemons)

Authorization Code Mode in detail

The most used OAuth 2.0 grant type is the authorization code. It serves as the best grant for web server applications and is implemented by both web and native applications.

Because the source code is hidden in a server-side web application, the confidentiality of the `"client_secret"` data may be maintained.

To obtain the access token from the authorization server, the client can set up a server-to-server connection. As a result, the token is not given to the user, preventing any potential unintentional disclosures.

Here is a high-level view of the main steps involved in this flow:

- The user is forwarded to the authorization server by the client application.
- The user verifies and accepts the application's request for authorization.
- The user is sent to the client after being given an authorization code.
- The client trades an access token for the authorization code.

The client application initiates the flow by requesting access to some user-owned private resources that are hosted by the resource server.

The resource owner is not required by the OAuth authorization protocol to divulge passwords to the client application. It is necessary for the user to authorize the client's request after authenticating it on the authorization server. The client consequently gets an authorization grant. The application establishes a connection with the authorization server and exchanges the received grant with an access token. Finally, the client can access the user's protected resources by using the access token.

Figure 2.2 contains a representation of the flow as it is described in the specification, from the request for permission to getting a legitimate access

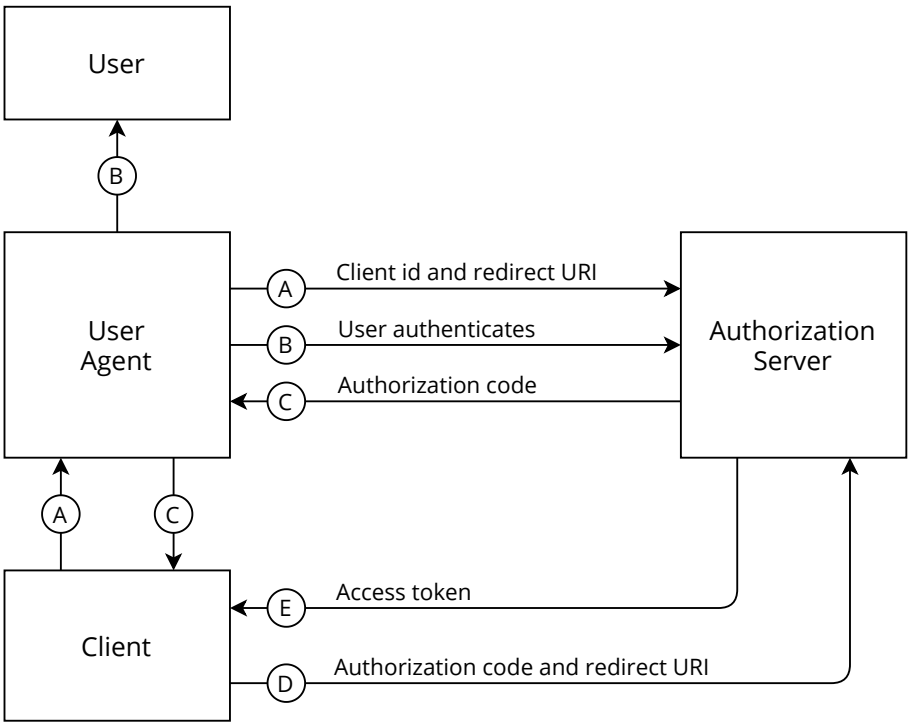

Figure 2.2: RFC 6749 - Representation of the authorization code flow

token. Due to the user-agent, the arrows for steps A, B, and C are split into two halves. We provide a detailed description of the protocol's steps as follows:

Authorization request The user is forwarded to the authorization endpoint by the client application. Two required parameters are included in the request URL: `response_type` and `client_id`. The value of the former, which indicates the grant type, must be equal to `"code"`. The public application identifier is contained in the `client_id` field and is given to the client by the authorization server upon the registration process.

```
GET {Authorization endpoint}
  ?response_type=code              // Required
  &client_id={Client identifier}   // Required
  &redirect_uri={Redirect URI}     // Optional
  &scope={List of scopes}          // Optional
```

```
  &state={Arbitrary string}        // Recommended
  HTTP/1.1
HOST: {Authorization server}
```

The location to which the user should be routed following the authorization stage is specified by the `redirect_uri` option. Typically, the authorization server receives the addresses used in the `redirect_uri` during the client registration process.

Using the `scope` parameter, the authorization endpoint's authorization request scopes can be specified by the client. The client application's access to the user's account is restricted through this approach. An infinite number of space-delimited strings, each representing separate permission, can be included in the `scope` parameter.

The `state` parameter contains a random string used by the application to prevent cross-site request forgery attacks. The value must be a secret and non-guessable nonce.

User authorization Following the authentication process, the authorization server shows the user an authorization form to authorize the requests made by the application. The user has the option to view a detailed list of all the permissions that the client application requires. If the user voluntarily agrees to the request, the authorization process is finished.

Authorization response The authorization server generates an authorization code for the client if all the request parameters are valid and the user has been correctly granted access. When the address indicated in `redirect_uri` is reached, the user is forwarded back to the client application. The authorization response contains the query parameters `code` and `state`.

The user will be redirected to the `redirect_uri` address if the authorization step fails, along with the additional `error` parameter in the URL to specify the reason for the failure. The OAuth specification includes a

comprehensive list of all error types.

```
HTTP/1.1 302 Found
Location: {Redirect URI}
  ?code={Authorization code}  // Required
  &state={Arbitrary string}   // Optional/Required
```

The client identifier and `redirect_uri` are tied to a temporary string that serves as the authorization code. If the authorization request included it, the answer must also include the `state` parameter. In this scenario, the client's original string must be returned. To mitigate any CSRF attacks, the program verifies the values' correspondence.

Access Token request The application can now obtain an access token due to a valid authorization code. The client sends a POST request, including the authorization code, the grant type `authorization_code`, and the redirection URI to the token endpoint of the authorization server.

According to the OAuth specification, in order to send requests to the token endpoint, the client must first authenticate with the authorization server. The HTTP Basic Authorization header can be used to handle the request authentication, with the parameters `client_id` and `client_secret` provided as the user name and password, respectively. The client credentials can also be added as additional POST parameters as an option.

```
POST {Token endpoint} HTTP/1.1
Host: {Authorization server}
Authorization: Basic {Client credentials}
Content-Type: application/x-www-form-urlecoded
  grant_type=authorization_code  // Required
  &code={Authorization code}     // Required
  &redirect_uri={Redirect URI}   // Optional/Required
  &client_id={Client identifier} // Optional/Required
```

Access Token response The authorization server verifies the client's identity and the information contained in the access token request. The authorization server issues an OAuth access token and transmits it to the client application in a JSON-encoded response if the validation is successful. The authorization server responds with an error message if the authentication is failed or the request is invalid.

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache
{
  "access_token": {Access token},       // Required
  "token_type": {Token type},           // Required
  "expires_in": {Lifetime in seconds}, // Recommended
  "refresh_token": {Refresh token},     // Optional
  "scope": {List of scopes}             // Optional/Required
}
```

The client application can utilize the access token obtained at the end of the flow to send valid API calls in order to access the user's protected resources.

Implicit Grant Mode in detail

The second grant type in the OAuth 2.0 specification is the implicit grant. It was created especially for native and JavaScript applications. The secrecy of the information included in the `client_secret` cannot be maintained because the source code of these types of applications is typically publicly disclosed. For this reason, the flow does not include any client authentication.

It was intended for the implicit permission to be more user-friendly than the authorization code. There is no intermediate code exchange phase in the flow, which is by its very nature straightforward.

The access token is shared with the user's browser in the authorization response rather than being sent directly to the client through a reliable server-to-server channel, which is a significant disadvantage of the implicit grant type.

This grant type hasn't been widely adopted yet because of the various security flaws discovered in several of the existing implementations of it. The implicit flow should never be used, according to the OAuth 2.0 Security Best Current Practice [74]. The value of the access token is vulnerable to serious leakage issues because it is included in the authorization response.

In any case, the implicit grant flow is secure. A faultless application that includes a compliant and comprehensive implementation of the protocol is likely to be secure. However, because of the flow's wide attack surface, it is rightfully regarded as insecure with regard to the code flow, and its use is still severely prohibited.

Figure 2.3 provides a graphic depiction of the stages taken during implicit flow. A and B's arrows are broken to show that they are moving via the user agent.

The user's browser is sent to the authorization endpoint by the client, starting the flow (A). The user verifies their identity on the authorization server and grants the client's requested permissions (B). The browser is sent to the client at the address supplied by the `redirect_uri` parameter if the authorization is successful.

The access token is contained in the fragment of the redirect URI (C). The browser sends a request to a client resource hosted on the web to complete the redirect action. The browser stores the fragment data locally, not sending it with the request (D).

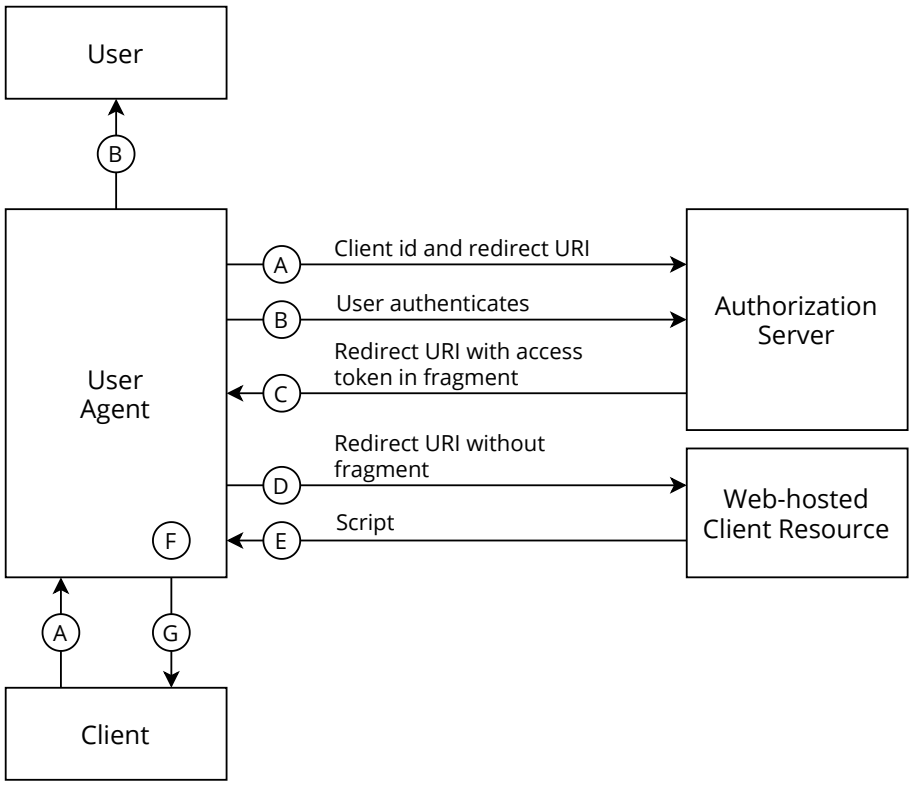

Figure 2.3: RFC 6749 - Representation of the implicit flow

A web page with a client-side script able to access the complete URI using the fragment is returned by the web-hosted resource (E). The script is run by the user-agent, and as it does so, it parses the fragment and determines the value of the texttt access token parameter (F). Finally, the browser provides the client with the obtained access token(G).

Authorization request

The client starts the process by sending the user's browser to the authorization endpoint. The authorization request provided for the authorization code flow is nearly identical. The `response_type` parameter's value of «`token`» is the only indication of the difference.

```
GET {Authorization Endpoint}
  ?response_type=token             // Required
  &client_id={Client identifier}  // Required
```

```
  &redirect_uri={Redirect URI}    // Optional
  &scope={List of scopes}         // Optional
  &state={Arbitrary string}       // Recommended
  HTTP/1.1
HOST: {Authorization Server}
```

The authorization server checks the request parameter values for accuracy and confirms that the redirect URI matches a previously registered address. If the checks are successful, the authorization server moves on to authenticating the resource owner, who must then grant the client's requested permissions. The user is presented with the same consent dialog as in the authorization code flow. The user's browser is then sent to the `redirect_uri` address by the authorization server.

Authorization response The client makes distinct requests to get the access token and the authorization code during the flow of the authorization code. The access token is instead given to the client directly in the authorization response in the implicit grant flow. The redirect URI's fragment component contains all of the parameters.

```
HTTP/1.1 302 Found
Location: {Redirect URI}
  #access_token={Access token}        // Required
  &token_type: {Token type},          // Required
  &expires_in: {Lifetime in seconds}, // Recommended
  &state: {Arbitrary string},         // Optional
  &scope: {List of scopes}            // Optional/Required
```

OpenIDConnect

It is a simple identity layer on top of the OAuth 2.0 protocol. It allows clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to gain the end-user's profile information in an interoperable and REST-like manner. Its current version is OpenID Connect which is the successor of OpenID 2 [90].

OpenID Connect performs many of the same tasks as OpenID 2.0 but does so in a way that is API-friendly and usable by native and mobile applications. OpenID Connect provides optional mechanisms for signing and encryption. Whereas integration of OAuth 1.0a and OpenID 2.0 required an extension, in OpenID Connect, OAuth 2.0 capabilities are integrated with the protocol itself.

OpenID Connect allows clients of all types, including Web-based, mobile, and JavaScript clients, to request and receive information about authenticated sessions and end-users. The specification suite is extensible, allowing participants to use optional features such as encryption of identity data, the discovery of OpenID Providers, and session management when it makes sense for them.[24]

2.2 Referrer Policy

The Referrer is an HTTP header field containing the address of the content from which the current resource has been requested. It allows the server to identify the URL of the resource from which the request was generated. A web application can use the Referrer field to generate back-links, to identify the location from which the requests come, for analysis and logging purposes [48].

Referrers may be required to properly configure tracking and advertisement systems. When a target content is obtained with a direct browser call from a source without a defined URL, the user agent does not include any Referrer field. Each content-initiated request is associated with a Referrer header. A referrer policy modifies the algorithm used to populate the Referrer when fetching sub-resources or performing navigation.

The empty string corresponds to the absence of a referrer policy. When a policy is not specified, the default value set by the browser is no-referrer-when-downgrade. With this default setting in place, the full URL is sent in the referrer field when the protocol security level is on the same level (HTTP $\rightarrow$ HTTP or HTTPS $\rightarrow$ HTTPS), and the field is left empty if the new request is directed toward a less secure destination[19].

2.2.1 Cross-Domain Referrer Leakage (CDRL)

When a web browser makes a request for a resource, it typically adds an HTTP header, called the "Referrer" header, indicating the URL of the resource from which the request originated. This occurs in numerous situations, for example, when a web page loads an image or script or when a user clicks on a link or submits a form.

If the resource being requested resides on a different domain, then the Referrer header is still generally included in the cross-domain request. If the

originating URL contains any sensitive information within its query string, such as a session token, then this information will be transmitted to the other domain. If the other domain is not fully trusted by the application, then this may lead to a security compromise. [43]

2.3 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 [96].

Since the SOP must be guaranteed by the client (e.g., browser) to assure the user's data integrity and confidentiality, it involves a large group of client-side scripting languages, such as JavaScript, which is implemented in almost all websites, even for a simple action. According to the SOP rule, the scripts from websites with the same origin can be run with no particular restrictions to access each other's methods, and resources [96].

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 in order to provide better functionality to users. To address this problem, some solutions have been introduced, such as postMessage and CORS.

The postMessage mechanism was first introduced by HTML5 for supporting cross-document communication in order to relax the same origin policy by allowing Web content from many origins to connect with one another. The implementation of postMessage typically involves two parties: the receiver and the sender. Every message has precise origin information, but the receiver must verify this information before receiving the message. Thus, the postMessage's receiver functions are given some authority to prevent cross-

origin attacks instead of the Web browser. [31]

In 2006, the W3C introduced another mechanism, called Cross-Origin Resource Sharing (CORS), to relax the SOP allowing cross-origin requests and sharing of resources between websites with different origins [40].

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" [118]. 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 [78].

```
OPTIONS /fold1/documents
Host: https://application.example.com
```

```
Origin: https://example.com
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: origin, x-requested-with
```

Before a browser makes any cross-origin request and it does not consider as a simple request, then the browser sends a preflight request to server. The server has to respond with the proper headers confirming to accept the request, and the browser will wait to send the real request afterwards.

In real-world scenarios, all cross-origin API requests will require the preflight requests with the following conditions:

- Requests with a JSON or XML body
- Requests with methods other than GET, POST or HEAD
- Requests including credentials
- Requests with any headers other than Accept, Accept-Language, Content-Language and Content-Type

Each of these requests prevents the original request from being processed for at least the round-trip time to the server. OPTIONS Requests are typically not handled by the CDN since they are not cacheable by default, therefore the server must be accessed each time. In many circumstances, this practically doubles the latency of the API for all browser clients. The performance has been cut in half from the perspective of the end user. It may significantly increase the demand on and cost of API servers. [16]

The "Access-Control-Max-Age header" can be used to optionally cache the preflight response for requests made using the same URL. The browser employs a unique cache for preflight answers that is distinct from the browser's standard HTTP cache. Preflight answers are never stored in the ordinary

HTTP cache of the browser. It is also crucial to set the 'Vary' header to 'origin' value because it instructs the cache, in addition to using the same URL, to use this response only for requests that have the same Origin header (requests from the same cross-origin source). [77]

CORS Security Analysis

Many developers, through different frameworks, generate dynamic CORS policies deployed in web applications and 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 any other domain and hinder SOP enforcement. Due to the policy's complexity and pitfalls in CORS design, there are several flaws in CORS implantation. The most common results of these mistakes in configurations are privacy leakage, information theft, and account hijacking. We provide the list of current faulty configurations in CORS, which are caused by validation mistakes or poor clarification of the policy for developers[40].

- **Reflecting arbitrary origin**: The basic way to configure CORS while dynamically generating the policies is to blindly reflect the "Origin" header value in the "Access-Control-Allow-Origin" headers in responses. This configuration could be dangerous since it trusts any website to read authenticated resources.

- **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., "attackvictim.com."

- **Not escaping '.'**: when the validation is performed using a regular expression, the developer might forget to escape the "." characters in the configuration. For instance, "victim.com" wants to allow "www.victim.com" but allows "wwwavictim.com" as well.

- **Value 'null'**: the "origin" header was first proposed as mitigation against CSRF attack, and CORS reuses this header [1]. 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. 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 could easily forge the "origin" header by sending a cross-origin request from an iframe sandbox in the browser. Consequently, a website with this flaw ("ACAO: null" and "ACAC: true") can be read by any other website.

- **HTTPS site trusts HTTP domain**: some CORS configurations do not take the protocol (scheme) into account and cause HTTPS websites to trust HTTP ones. In this situation, a network attacker could compromise the user's communication with the target website, steal their sensitive information, and hijack HTTP content to perform a CSRF attack against HTTPS.

- **Arbitrary subdomain**: most applications decide to allow access from all their subdomains (even non-existent ones). Additionally, some websites allow access from various third-party websites, including all their subdomains. This kind of mistake in CORS implementations can lead to unauthorized access from arbitrary subdomains by related-domain attackers.

2.4 Cross-Site Attacks

Since browsers attach cookies to all requests by default, they allow an adversary to deliver malicious payloads through user authentication.

A large part of web security is involved in cross-site (XS) attacks, in which victims visit a malicious website that sends cross-site authenticated HTTP requests to a vulnerable website, tricking their browser. The malicious website references the resources of the vulnerable website through a hidden HTML form or JavaScript code.

When the user visits the malicious website, the browser automatically attaches the user's authenticated session cookies to the request, and the vulnerable website processes the request, providing an authenticated response [66]. Cross-Site Request Forgery (CSRF) and Cross-Site Scripting (XSS) are some examples of cross-site attacks.

2.4.1 CSRF

Authenticated users are made to take unwanted activities on a web application via a technique known as cross-site request forgery (CSRF). The attacker deceives the victim in order to execute specially crafted malicious requests to a target website.

The trust relationship between the web application and the browser is exploited by the attack. The victim's browser includes the authentication cookies in all requests made to the target website if the user has been authenticated there. [30]

The web application is unable to distinguish between genuine requests and malicious requests that are generated by the user's browser but are forged by the attacker. State-changing activities, such as modifying the victim's login information or completing transactions, are the attack's main objectives. [120]

Login CSRF

A login cross-site request forgery occurs when the attacker tricks the victim into sending a cross-site request to the target website's login endpoint. The login request is forged by the attacker using its own credentials. If the attack is successful, the server sends a session cookie to the victim's browser.

As a result, the victim is logged into the target website with the account of the attacker [30]. There has been several studies [70, 125, 98, 107, 27] analysing the login CSRF attacks.

The attack could seem quite harmless at first glance. An attack using cross-site request forgery often targets actions taken on the victim's protected resources. The attacker uses a login CSRF to trick users into carrying out undesired actions inside the attacker's account by taking advantage of an application bug. After the attack is carried out, the browser's state is altered, and the victims might not even be aware that they are logged into someone else's account. As a result, individuals might upload delicate files, divulge credit card information, and share other private information with evil users. [30]

Impact: Some web applications let users register multiple OAuth provider logins that are connected to the main account. It stands for an alternative and less complicated method of using the application. In this scenario, a login CSRF attack could result in account takeover if one of the implemented OAuth flows is unsafe. The ability to link accounts can be used to access the victim's data completely.

The security researcher Egor Homakov initially outlined the attack flow in an essay on his blog, [60]. Only if specific requirements are met is the attack feasible. The target website must use at least one flow implementation that is vulnerable in external login procedures based on OAuth 2.0 in order to be vulnerable. Only users that have been authenticated by the target client

application are eligible to receive the attack.

With the target website's identity provider using a susceptible implementation, the attacker starts the authorization process. Following the server's authorization response, the OAuth flow is immediately blocked. The attack vector is represented by the retrieved redirect address. The victim is persuaded to click on the link or visit a malicious website that contains it.

In the latter case, the malicious request often starts without any additional user input. All of the remaining OAuth flow steps are completed on the victim's own browser once the victim opens the authorization response. The attacker's account is connected to the victim's account at the conclusion of the attack. As a result, using the identity provider's profile used in the attack, the attacker can access the victim's account on the client application. [25]

Mitigation against CSRF attacks

There are some mechanisms that a site can use to defend against CSRF attacks [87]; verifying the origin with standard headers, synchronizer token pattern, SameSite Cookie Attribute, double submit cookies, and use of custom request headers, to name but a few.

As we study CSRF in Single Sign-On through a web attacker, we consider the three effective defense mechanisms in this manner: validating a CSRF token to all state-changing requests(`state` parameter, in case of SSO), including additional headers with XMLHttpRequest, and user interaction as these three defense mechanisms were used repeatedly throughout our large dataset.

These mechanisms are in use on the web today, yet they are not entirely satisfactory. However, we observed a small fraction of cases using other defense mechanisms which were effective in defending against some of the proposed attack scenarios.

State parameter The demonstrated CSRF attacks against OAuth are

successful because the weak client application fails to appropriately confirm that the user who submitted the authorization request is the same user who is submitting the response. This check can be performed in the OAuth 2.0 flow with the analysis of a security parameter named `state`. [32]

According to OAuth 2.0 specification [58], for its redirection URI, the client must implement CSRF protection. A value that ties to the user-authenticated agent's state must be part of every request delivered to the redirection endpoint.

> «The client SHOULD utilize the `state` parameter to deliver this value to the authorization server when making an authorization request. The authorization server must redirect the user-agent back to the client with the required binding value contained in the `state` parameter. The binding value enables the client to verify the validity of the request by matching the binding value to the user-agent's authenticated state.»

[112]

This value `state` parameter which can be performed in OAuth 2.0 flow. The state parameter, to prevent CSRF attacks, should be a non-guessable randomly generated sequence of characters.

> «The authorization server MUST prevent attackers from guessing generated tokens, the probability MUST be less than or equal to 2^{-128} and SHOULD be less than or equal to 2^{-160}.»

[58]

The client's security against a CSRF attack is not assured by the presence of the parameter, though. The vulnerability of the application may result from incorrect validation or improper processing of the parameter, as seen in [125].

Custom HTTP Headers Sites implementing AJAX interfaces can defend against CSRF by setting a custom header via XMLHttpRequest (e.g., X-Requested-With) to all state-modifying requests. It has been suggested that the presence of the X-Requested-With header ensures that the request originated from a trusted domain.

For example, to defend against login CSRF, the site must send the user's authentication credentials to the server via XMLHttpRequest. In this way, the site validates the headers that guarantee the integrity of the request and rejects all state-modifying requests that are not accompanied by the header. [107] Since, in this research, none of the attackers have control over the origin of the websites, custom HTTP headers are an effective protection mechanism against CSRF attacks.

However, custom headers can only be used by websites implementing all their security-sensitive operations via JavaScript. In our experiment, we show that despite implementing this custom header, there are websites that are still vulnerable to Login CSRF.

User Interaction The proposed defenses against CSRF attacks do not require any user interactions. However, a crucial feature of CSRF is that web applications cannot recognize whether or not a request is intentionally issued by the user, So involving the user in performing an additional step before accepting the request could prevent unauthorized operations.

Examples of these steps that can act as strong CSRF defense when implemented correctly are as follows: 1) user Re-authentication, 2) One-time Token, and 3) solving CAPTCHA challenges. Even though this type of defense is very effective against CSRF, it can create an unpleasant impact on the user experience, and for this reason, only a small fraction of websites implement it, as shown in our result in section 4.4.2.

2.5 Cookies

Modern web apps continue to be developed on the foundation of simple web primitives built for less complex applications, even though they now offer rich user experiences that can compete with those of desktop applications.

In particular, HTTP, a stateless protocol, transports messages between the user's browser and the server. Since practically every application requires a state that is unique to each user, the server must explicitly connect these messages into "sessions" in order for this data to be accessed and changed throughout a user's interaction with the application. There are several techniques for maintaining this session state on the web, and the most commonly used ones are Cookies. [34]

Cookies are a primary HTTP mechanism to secure the session's integrity and confidentiality. The consequences of compromising the confidentiality or integrity of a user's session are critical: stealing the sensitive data, hijacking the account, and replacing the session (e.g., login CSRF [32]).

To do so, cookies allow websites to store key/value pairs using the HTTP Set-Cookie header, and then the user agent returns them on subsequent requests. One of the attributes is SameSite handling the restriction of the user's session to a first-party or third-party request.

SameSite

Generally, cross-site attacks are successful when the browser includes valid cookies in all the requests; therefore, an effective solution to cross-site attacks is to set restrictions on cookies' scope, instructing the browsers on whether to include them in the outgoing requests using the SameSite attribute in the Set-Cookie HTTP response header. The SameSite attribute introduces three cookie policies: None, Lax, and Strict [66].

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.

Lax The Lax policy tries to increase the usability of the website and yet its security, maintaining the user's logged-in session when the user arrives from an external link. In fact, 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 Chromium-based browsers, while Firefox and Safari default the attribute to None.

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.

Secure

The Secure attribute of cookies allows for restriction of the use of cookies to secure connections, i.e., generally HTTPS connections. This attribute only protects the confidentiality of the cookies, as a network attacker might still break their integrity, rewriting their value by injecting a Set-Cookie header into a response to the victim [29].

2.6 Web Cache

The cache lies in between the user and the server and acts as man-in-the-middle proxy devices. The Cache interface provides a persistent storage method for Request/Response object pairs cached in long-term memory. It stores (caches) responses to specific requests, typically for a specified amount

of time. When the next user makes the same request, the cache simply delivers the user a copy of the previously cached response without involving the back-end in any way. By lowering the volume of duplicate requests the server must process, this significantly lessens the server's workload[7].

The implementation of web caches occurs at many points along the traffic delivery channel, starting with the private caches found inside browsers and ending with the application caches set up alongside the origin server, including any caching proxies that may be present in between. First and foremost, Content Delivery Networks (CDNs) with their extensive networks of caching proxies (also known as edge servers) have gained widespread adoption. [15, 17]

Website owners have a wide range of options with CDNs to modify the caching behavior to suit their requirements. The request endpoint, file extension, query string arguments, the presence of cookies, the request headers, the response content type, or a complex combination of several such attributes, for instance, can all be used to determine whether to cache a response. Major CDNs have more recently begun to provide edge computation capabilities, allowing website owners to automate these decisions. An origin can have multiple, named Cache objects. The vendors must handles Cache updates; items in a Cache are not automatically updated until specifically requested, and they do not depreciate unless removed. [18, 54, 16, 53]

Web Cache Poisoning

Web cache poisoning was first introduced in Practical Web Cache Poisoning research paper [9] in 2018. In this technique, an attacker manipulates a web server's and cache's functionality to deliver a malicious HTTP response to other users. A poisoned web cache could to be severe to other users by spreading several different threats, using vulnerabilities such as XSS, JavaScript injection, open redirection [10].

Only users who access the affected page while the cache is poisoned will receive the poisoned response and depending on the popularity of the website, its impact might be massive. Web cache poisoning essentially comprises two stages. The attacker must first figure out a way to get the back-end server to respond with an unintended harmful payload. Once they are successful, they must make sure that their response is stored and then served to the target audience.[84]

Chapter 3

Related Works

In this chapter, we include the related works on both OAuth protocol security and Cross-origin communication security. We mention the works on OAuth security in general. However, since we consider CSRF attacks in OAuth in our research, we illustrate the studies mostly related to OAuth CSRF in detail. For cross-origin policy, we mention another mechanism, called postmessages, other than cross-origin resource sharing (CORS). However, we describe mostly the works related to CORS in detail.

3.1 OAuth

Several formal approaches have been used to examine the OAuth 2.0 protocol, which involves an examination of the protocol flow.

Although protocol analysis techniques are useful to verify the security of protocols, they assume websites are correctly implemented, and the implementation details and browser behaviors might be ignored. Hence some other researchers performed a security analysis through the real-world OAuth implementations.

3.1.1 Formal Approaches

Chari *et al.* [38] analyzed the security of the authorization code mode in the universally composability [37] security framework. They defined an ideal functionality for OAuth protocol and implemented it without considering the web features, i.e., the semantics of HTTP status codes or details of cookies.

With the use of formal methods and the verification of the associated flows and state changes, Pai et al. [88] examine the theoretical security of the protocol.

To define OAuth 2.0 with all of its restrictions and behaviors, they employ the Alloy modeling language and thanks to this intended model, an OAuth client credentials flow security hole was found. To assess the security of the OAuth protocol in other works, abstract models were utilized, and they demonstrated how the finite-state Alloy checker was able to identify the known OAuth flaws. However, the theoretical method has the drawback of not allowing for the discovery of implementation-related problems. [67]

Wang *et al.* [115] present a systematic approach to finding implicit assumptions in SDKs used for authentication and authorization, including SDKs that implement OAuth 2.0.

In their formalization of OAuth 2.0, Bansal *et al.* [28] perform a formal

analysis of OAuth 2.0 to discover attacks on OAuth. They use the WebSpi library [14], which defines the basic components to model web applications and their security policies, ProVerif [33], and a description of new concrete website attacks found and confirmed by their formal analysis. They identify previously unknown attacks on the OAuth implementations,i.e., and Social CSRF attack, based on a customized attacker model and finer-grained web security mechanisms.

Another formalization by Fett *et al.* [47] perform the extensive formal analysis of the OAuth 2.0 standard for all four modes, which can even run within the same and different clients and APIs, based on a comprehensive and expressive web model, covering the part of how browsers and servers interact in real-world configuration.

While proving the security of OAuth in the web model, they discovered four new attacks which break the security of OAuth: 307 redirect[81], Mix-up, State leak, and Naive client session integrity. They proposed some solutions for recognized vulnerabilities to fix them.

3.1.2 Empirical studies

Many empirical works have been done on the security of OAuth-SSO (e.g., [82, 46, 114, 26, 111, 110, 44, 99, 22, 106]) either by developing web-based tools or evaluating the risk in real-world implementations. In the following, we illustrate the papers covering OAuth and CSRF attacks. We categorize these related works into three groups: mitigation, protection tools, and security analysis.

Mitigation

There are some researches focusing on mitigations against CSRF(e.g. [71, 49, 20, 73, 97, 116, 13]).

Li et al. [71] involve the analysis and validating Referer header field performed by the relying party. The technique allows preventing the execution of OCSRF initiated from domains under the control of an attacker. They propose an attack model and how this mitigation is effective in some cases. However, in our paper, we test a larger portion of websites automatically, including more mitigations in our results.

In [73], the authors provide a broad security evaluation of CSRF mitigations throughout the popular web frameworks. By dissecting CSRF attacks, they identify 16 existing defense mechanisms from literature and non-literature resources, which are considered in four distinct categories; Origin checks, Request Unguessability, SOP for Cookies, and User Intention.

They discover a few vulnerabilities in some of the web frameworks which allow the attacker to bypass the CSRF defense. They show that even though CSRF defenses have been implemented, much of their correct and secure implementation depends on developers' awareness of CSRF attacks and specific behaviors of the implementations. However, they analyze and evaluate the frameworks, web languages, and current documentation to assess the CSRF attack in general. Unlike our research, not only is their evaluation not fully automated, but also they do not consider all the different factors in the real scenarios which the victim might face in the wild.

Protection tools

Another thread of research [35, 72, 127, 123, 121] focuses on designing and implementing browser-side tools to protect users against web attacks regarding the OAuth protocol.

The user's browser will attach the cookies to the request that the malicious code sends to the target web application. This form of behavior could be exploited by CSRF attacks. There are some works suggesting to modify this behavior of the browser, via proxies (e.g., [42, 22, 64]) and browser extensions

[63, 75, 93, 94, 68]

The authors in [121] implement a protection tool providing security to vulnerable web APIs communications. It works as a proxy on the service's website, checking on the set of invariant relations among the HTTP requests and responses during an SSO action. They show that their tool is effective in complicated situations and defeats the exploits on high-profile web applications.

Zhou and Evans [127] developed SSOScan, an automated vulnerability scanner, which was used to check the security of Facebook OAuth 2.0 implementations in 1,660 heavily frequented websites. The tool was made to involve humans as little as possible.

The Facebook button is located inside the login page using a module called SSO Button Finder, which automates the permission procedure. In our research, a similar keyword-based approach was used to successfully automate the information retrieval phase. Our crawler's development was greatly influenced by additional features of their implementation.

Calzavara et al. [35] design and implement a browser-side security monitor for web protocols, called WPSE, to prevent nine attacks violating the security properties of OAuth. However, WPSE cannot prevent certain classes of attacks, including automatic login CSRF attacks, network attacks that are not observable by the browser, and impersonation attacks.

The authors in [72] develop OAuthGuard, a real-time vulnerability analyzer, and protector, and put it into use as a JavaScript Chrome browser extension. Implementations of OAuth 2.0 and OpenID Connect can both be secured using it.

OAuth 2.0 Detector, Vulnerability Analyzer, and Vulnerability Protector are the three primary parts of the program. The OAuth protocol-related requests are recognized using the first module. The vulnerability analyzer examines network traffic to look for vulnerabilities, and it enables the detec-

tion of CSRF attacks against the redirect-uri in addition to other security risks.

Yang et al. [123] develop S3KVetter, an automated testing device, to confirm the SSO SDKs' implementations' security. An SDK's source code is frequently made available to the public, enabling white box testing methods to be applied to the distributed libraries.

Dynamic symbolic execution was used to analyze each SDK's source code. The tool tests the interactions between the players engaged in the flow and handles the multi-party aspect of the protocol correctly. S3KVetter identifies seven categories of logical errors.

Security Analysis

There are some papers with an approach similar to ours [70, 108, 125, 98, 107]. In the following, we provide a detailed explanation of these works and the gaps we address in this paper.

Sun and Beznosov [110] examine the Facebook SSO implementations on 96 client websites as well as the implementations of three OAuth 2.0 identity providers: Microsoft and Google.

They examine the network traffic produced by the user's access while treating the relying parties and authorization servers alike as black boxes. A subset of 15 depending parties was queried manually for traffic, and the data was then semi-automatically analyzed using some established tools. The information extracted from the network traces led to the discovery of various known security flaws.

Li and Mitchell [70] analyze the security of OAuth 2.0-based SSO implementations in 10 identity providers and 60 Chinese-reliant parties. The work is concentrated on CSRF attacks using OAuth against websites using identity providers. To gather and examine the network traffic generated while the protocol was being executed, they deployed a proxy.

A sizable number of case studies were produced once the discovered security flaws were manually examined. To make traffic analysis easier and cut down on errors caused by manual inspections, Java software was created.

They discovered that a sizeable percentage of clients were not using any defenses against the CSRF attack against the redirect-uri. Many of the websites inspected did not include a `state` parameter in the OAuth flow or used a constant value, and some relying parties failed to bind the `state` with the user's session.

Sumongkayothin et al. developed OVERSCAN [108], a Java security scanner that can spot missing or improperly utilized parameters in the OAuth 2.0 protocol. The utility was developed as an add-on for the Burp Suite software. It functions as a proxy and gathers and examines network traffic flowing between the browser and the target web applications.

Only the requests related to the OAuth protocol are examined; the classification is based on the presence of the `response_type` variable in the query string. OVERSCAN can detect the absence of the `state` value in the OAuth authorization request, among other parameters. The severity level of this flow misconfiguration is considered high. The researchers used the security scanner to conduct an experimental examination. The results revealed that in 5 out of 45 web applications, the `state` parameter was missing.

Yang et al. [125] developed OAuthTester, a model-based method for automatically finding flaws in OAuth 2.0 implementations. OAuthTester builds a state machine from the protocol specifications in order to overcome the shortcomings of earlier theoretical approaches, but it then improves the state machine and fills in the blanks left by the ambiguities of the specification by tracking the traffic of the OAuth flow and the server state.

However, as the authors note, they can only examine HTTP traffic. Thus they are unable to obtain all of the information we utilized to develop the attack tactics. As a result, they miss the vulnerabilities that our analysis

identifies.

Shernan et al. [98] conducted a large-scale analysis to determine whether CSRF vulnerabilities existed in actual protocol deployments. They implemented a lightweight crawler named OAuth Detector (OAD), a high-performance tool specifically developed to gather and analyze OAuth-related traffic in a sizable number of well-known websites.

A review of the top 10K Alexa domains indicated that 25% of OAuth-using websites were susceptible to CSRF attacks. This outcome can be explained by the frequent lack of fundamental safeguards against CSRF vulnerabilities. The OAuth 2.0 protocol makes it very explicit that in order to protect the flow, some kind of CSRF mitigation must be used to secure the flow.

However, several of the major identity providers examined in this paper either suggested using the `state` parameter in the flow or outright disregarded it. The OAD tool performs crawling at a depth level of two, beginning on the home page of each website.

In order to find the requests connected to the OAuth authorization code flow, the acquired links are examined. Any website that contains an exposed request is marked as vulnerable. The approach to determining vulnerability is based on the absence of the `state` variable in the OAuth authorization request. Although the inspection is simple to do, unfortunately, it leads to partially unreliable results.

Sudhodanan et al.[107] present a comprehensive study on the different types of authentication CSRF reported in the literature. For identification of strategies in order to detect and reproduce each vulnerability, they use the same browser to simulate the interaction between the attacker and the victim, which led to missing some additional scenarios regarding the victim's browser states at the time of the attack.

Our approach considers additional attack strategies. Instead of using the same browser, we totally separate the environment in which the attacker and

victim operate. In place of performing the attack in a clean browser session, we also perform tests in the presence of a visitor and authorized cookies, which are not considered in their analysis.

Compared to related works, we identified three different types of login links possible through OAuth protocol (direct link, internal link, and button link), and we carefully implemented and automated the login part. In addition, our tool does not have the limitation of language or location of the website.

3.2 CORS

The SOP is too restrictive for the modern Web and the W3C standardized two new mechanisms of cross-origin communication, namely postMessages and CORS, in order to relax SOP.

There are many studies (e.g.,[101, 104, 122, 52, 103]) aiming at the security issues in postMessages mechanism and analyzing their incorrect implementations in the wild.

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 ([21, 83, 23, 65, 61, 55, 91, 119, 128]).

In [128], the author analyzes the security risks of applying CORS in local storage. By considering some concrete attack cases, they propose a practical scheme and some recommendations for the safe use of CORS in Local storage.

Gurt [55] found a CORS configuration mistake in one of Facebook Message domains that caused the exposure of the victim's private information in the chat box by any malicious website.

Revay [91] found a file upload CSRF vulnerability that was caused by a POST body format that was relaxed in XMLHttpRequest API.

Wilander on Github [119] show that an attacker may deliver malicious payloads via the three headers (i.e,`Accept`, `Accept-Language`, and `Content-Language`) to. They suggested that the Fetch standard should restrict these headers' values according to RFC 7231 [62].

The authors in [126] and [100] analyze inconsistent access control policies for different resources in web browsers (Cookies and DOM), but they did not include CORS.

Kettle [65] provides a summary of several CORS flaws identified throughout his penetration testing experiences. Müller [83] 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 [61], showing a high number of vulnerable websites in the Alexa top 1M.

Chen et al. [40] 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.

Meiser et al. in [78] 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. Based on this graph, they estimate the damage of XSS exploitation that usually occurs when websites trust each other in the interconnected web. As for CORS, they only investigate the explicit trust that enabled it, disregarding flaws.

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.

Chapter 4

OAuth

Many works have been done on the security of OAuth protocol, whether formal or empirical, and OAuth specification has been published, providing the best practice for developers and vendors. With all, significant fractions of websites are still vulnerable, not only because of implementation mistakes but also because of other factors unrelated to the application (e.g., cookie policy on modern browsers). In the following chapter, we provide the automated methodology targeting many web applications regarding their language or locations and implement the most comprehensive attack scenarios.

4.1 Motivation

At the beginning of the study on OAuth, we investigated the OAuth-based SSO implementation on top 5k domains since OAuth-based SSO implementation is prevalent in the wild.

Since the study was manual, the registration and login part was time-consuming, and we only did it on 160 websites at the time. We conducted a measurement study on the captured traffic and characterized Cross-Domain referrer Leakage (CDRL) vulnerabilities in OAuth implementation.

We perform a custom web crawler to collect the data used in this measurement. The crawler was configured to identify vulnerable websites. Each link crawled, and all generated requests by the browser were captured. One hundred sixty websites generated a total number of 6,824 unique HTTP requests by the simulated browser.

After inspecting these requests, 4,195 requests exposed sensitive data (such as token, username, and state parameter) in the referrer field. However, other requests do not contain the state parameter or access token in the referrer and, for this reason, cannot be considered vulnerable.

Regarding the analysis of the collected data, a large fraction of websites that implement OAuth protocol suffers from a serious web vulnerability (CDRL) which causes an unwanted leakage to untrusted third parties. In one scenario, if a third party is compromised, many state parameters and access tokens would be disclosed to an attacker.

Even though CDRL is a serious vulnerability, we aim more dangerous one in OAuth protocol: Cross-Origin Request Forgery. We learned our lesson from this primary study to achieve better results: automated and repetitive methodology to cover more and more websites. Much research has been done on CSRF attack in OAuth, so in the following, we provide a comparison within all the current studies related to OAuth CSRF attack, hence our

motivation for this part of my thesis.

4.1.1 CSRF

In our work, we proposed an automated technique to perform a large-scale analysis of a large number of popular websites. The purpose of the study is the identification of vulnerable OAuth 2.0 implementations related to the identity services of Facebook and Google.

As reported by [107] CSRF vulnerabilities related to authentication and identity management services are extremely pervasive, even among the top-ranked websites.

Our analysis is aimed to discover the incidence of the attack in the wild under different conditions. We designed an automated testing framework to programmatically perform this evaluation.

Other works tested the attack only within a clean browser session, but this setup does not accurately represent a real attack scenario.

At the time of the attack, the victim may already be logged into the website, and the browser may have set some cookies. The presence of visitor or authentication cookies can significantly influence the outcome of the attacks.

Our study is aimed to gain new knowledge to understand the impact and the outcome of the CSRF attack in scenarios resembling real-world situations which lead to serious consequences, ranging from the disclosure of sensitive information to a malicious user [30] to the complete account takeover [60].

As a minor consideration, we worked with a heterogeneous set of websites with different languages and alphabets located in different countries. Our dataset contains, for example, Japanese and Russian websites, as well as websites registered in the United States and in several European countries.

Our approach overcomes the limitations of other works [70, 110], which only analyze websites providing a user experience in a specific language.

The main limitation of the manual approaches is that they represent a

small number of websites that can be examined. The lack of automation makes the inspection process extremely time-consuming, and the restricted size of the considered sample makes it difficult to discover vulnerabilities and error patterns.

This approach is suitable for examining the details of a specific vulnerability but is inadequate for large-scale security evaluations. To obtain statistics on the prevalence of OAuth vulnerabilities in the wild, it is required to overcome and extend the manual approach. For example, a higher level of automation can be achieved with the use of crawlers and software to control and simulate browser navigation.

With the aim of estimating the incidence and real impact of OAuth vulnerabilities in a heterogeneous world of applications, several large-scale measurement studies have been performed over time.

Our tool considers many more implementation parameters and factors present in real-world implementations than existing tools, and it does not remove any website (because of the presence of state parameters).

The tool works with a larger number (15) of different attack scenarios compared to the existing tool; some of these scenarios are completely new. The scenarios are all automated and implemented without user interactions, which are assessed to find more vulnerable websites. Unlike previous works, we analyze the impact of these attack scenarios on other CSRF mitigations and find that these factors could easily affect and disable other mitigations.

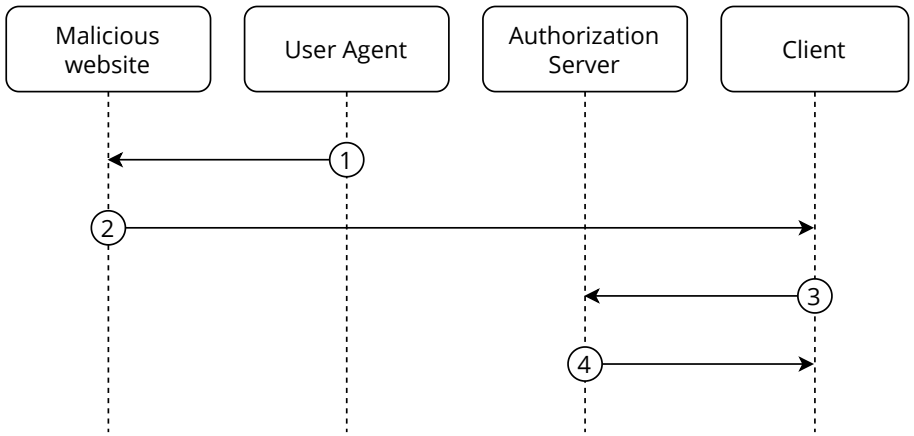

Figure 4.1: The main steps involved in the CSRF attack against the redirect-uri

4.2 Threat Models

The redirect URI used in the authorization code and implicit flows is targeted by the attack, which is defined in RFC 6819.

After getting an authorization code or an access token from the identity provider, the attacker starts the authorization process to access his own protected resources and terminates it after. The victim receives the retrieved URL and is deceived into executing the redirect back to the client application.

In case the authorization code flow is used, the client application generates a server request to exchange the authorization code for an access token.

In the implicit flow, a client-side script that is run by the victim's browser retrieves the access token from the URI fragment and sends it to the client. In both situations, the victim may utilize the access token given to them by the identity provider at the end of the OAuth flow to access resources on the attacker's behalf.

The login CSRF attack described in the preceding paragraph shares the same risks and impacts as the CSRF attack against the redirect-uri.

Figure 4.1 outlines the key attack steps within the context of the authorization code flow. In the illustrated example, the attacker stored a link leading to the redirect URI of the target website on a malicious website. The

victim's browser opens a malicious website to begin the operation (1).

The browser typically launches the attacker's specially constructed request automatically as the page loads (2).

The victim's side then completes the OAuth flow that the attacker started. The authorization server's token endpoint receives a request from the client application. The request contains the attacker's authorization code generated by the AS (3).

The identity provider returns the client's access token after exchanging the received code for one (4). The client can now use the token that was just received to get the data required to authenticate the user in a case that OAuth. The login is carried out using the attacker's account because the attacker started the flow.

4.2.1 Enabling factors

In a practical setting, a number of variables can affect how the CSRF attack against redirect-uri turns out. What takes place if the victim is logged into the vulnerable application? Does it matter if a person has never been to the website before? Is the attack stopped if the victim has already verified on the website? Understanding the effects of OCSRF in practical situations requires the answers to these issues.

The victim does not have to be authenticated on the target application for the CSRF attack against the redirect-uri to be exploitable. The attack frequently succeeds even if the victim has never been to the website before. However, the presence of cookies that the target website had already set in the browser can change how the attack turns out.

In our analysis, we investigated this hypothesis by running all the test scenarios with three different configurations:

(a) No cookies

(b) Visitor cookies

(c) Authentication cookies

The attacker's forged link is opened in a new browser session in the simplest configuration, indicated by the letter «a». The browser instance is clean, and there are no cookies set by the client application. In order to simulate a situation when a user is attacked who has never visited the website before, this parameter is employed. As soon as the browser is launched, the victim clicks on the malicious link.

In the second configuration, the target website's login page is accessed before the attack is carried out. The victim's browser starts the login process, and the crawler is then forwarded to the IdP login endpoint. On the target website, the navigation causes the creation of some visitor cookies. The attack is then carried out. The victim had previously visited the tested website before the attack, which is represented by the configuration's name, «b».

The victim is first authenticated on the target site in the final configuration, known as «c» before the attack is performed. With the victim's account, the crawler logs into Facebook and then checks the target site's login status. The attack is launched at this point. The interaction between two authenticated actors makes this scenario the most difficult to accomplish. We employed two registered users—one as the attacker and the other as the victim — to fully automate the testing process.

4.3 Methodology

We designed a repeatable methodology to discover and validate OCSRF vulnerabilities in targeted websites. As depicted in Figure 4.3, our methodology has three phases: 1. target selection, 2. measurement setup, and 3. OCSRF detection. We developed a tool based on Python-Seleniumto automatically select targets and test different OCSRF scenarios.

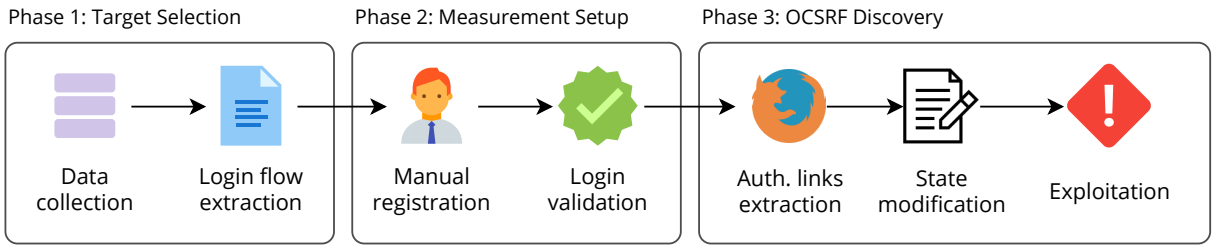

Figure 4.2: Abstract view of OCSRF detection methodology.

4.3.1 Phase 1: Target Selection

Step 1: OAuth Login Detection For extracting the initial seed set of candidate websites using OAuth login, we developed a browser-based crawler to visit websites in the initial seed set (e.g., Alexa Top 50K) in April 2020. The crawler is designed in a way that extracts initial OAuth login links for specific popular providers via checking the presence of OAuth standard parameters in all extracted links.: `response_type`, `client_id`, and `oauth`. The string «`oauth`» is commonly contained in the URL of authorization endpoints, and its presence is a good indicator of the existence of an OAuth-based process. All these parameters are used by the crawler in the detection phase to classify the links and identify the different login systems built on top of the OAuth protocol. Since many websites use JavaScript, which requires interaction with users to trigger OAuth login, we develop a browser-based crawler to increase the detection rate.

Step 2: OAuth Flow Extraction In order to remove false positives and extract OAuth redirection flows properly, the crawler follows all extracted and selected links. If the crawler lands on any well-known identity provider, we will add the site to our candidate list, which will later be used to test our OCSRF attack strategies. A keyword-based approach is used to detect the Login/Sign-in buttons (these elements usually contain some known keywords to identify the login action and the identity provider). Extracted flows would later be fed into the next phases.

4.3.2 Phase 2: Measurement Setup

Step 1: Manual Registration We follow the extracted OAuth links and create two sets of test accounts (victim and attacker) for each targeted site. Since the information provided by the external identity provider is not sufficient for the account creation process in many targeted websites, manual data entry is necessary. We adopt the previously proposed technique[79] to populate attacker and victim accounts with unique information (e.g., name, email, user identifier, phone number, profile logo, etc.) and use them in the next steps as `markers`.

Step 2: Login Validation To verify the login steps, the crawler uses the login information gathered in the first phase to initiate the OAuth login trail. It reaches the authentication page and enters the credential automatically. At this point, the flow is complete, and the browser is redirected to the target site's landing page.

OCSRF attack detection requires a victim to log in as an attacker to the targeted site. The detection crawler should be capable of detecting the forged login to the attacker's account. In this regard, a learning process is developed for the crawler to automatically complete and learn the login processes for both attacker and user accounts. In the learning process, the crawler scans the HTML code of the landing page and looks for specific user-related strings.

We presume the presence of some predefined unique `markers`, visible only as a result of a valid login to each account (which is populated to each account in the registration step).

4.3.3 Phase 3: OCSRF Discovery

The main goal of this phase is to discover exploitable websites. The crawler is designed in a way to discover various implementation flaws in `state` validation (described in step 2). In the first step, the crawler follows the OAuth flow, logs into the attacker account, and extracts the authorization response links. In the second step, the crawler applies different modifications based on five attack strategies on the extracted authorization link. In the last step, the different victim browser status is exploited with modified links.

Step 1: Authorization link Extraction Since the successful exploitation of OCSRF needs an attacker authorization response link including authorization code, `state` etc., the crawler initially follows OAuth login and obtains an attacker authorization response from the identity provider. We develop a browser extension to allow the crawler to record the attacker authorization link from the identity provider and halt the OAuth flow immediately. In other words, the generated authorization link is recorded, and the OAuth flow is stopped before redirection to the target site. The extracted link will be modified in the next steps to discover vulnerable websites.

Step 2: `state` Modification

The extracted authorization link would be modified by going through five attack strategies. All attack strategies are performed mainly based on modifications on `state`, as a result of which attack URLs would be created. The first scenario is applied to the subset of websites in which a `state` is not present in the authorization link. In other scenarios, the attack strategy would build further attack URLs by manipulating the `state` value as enumerated as follows.

0. **No `state`.** The link is sent unaltered to the victim if the original link does not contain a `state`.
1. **Empty `state`.** The `state` value is replaced with an empty string.
2. **Lack of `state` validation.** The value of the `state` is replaced with a randomly generated string.
3. **Unlinked `state`.** The link including `state` is sent unaltered to the victim.
4. **Missing `state`.** The `state` is removed.

In the first attack strategy, the authorization response link obtained at the first stage remains unchanged. In order to build other test cases, the testing strategy would manipulate the value of `state` value by either replacing it with an empty string, substituting it with a randomly generated string, or keeping the same value. The last attack strategy would completely remove the `state` parameter. In both strategies 0 and 3, the attacker would deliver the attack link unchanged to the victim. The strategies 1 and 2 rely on different alterations of the `state` value. In strategy 1, the content of the parameter is replaced with an empty string, while attack strategy 2 replaces the value with a random string. Finally, in the last strategy, the `state` value and parameter name are completely removed.

Step 3: Exploitation Each of the attack URLs generated in the previous step would be opened in a separate browser. We propose several OCSRF test cases based on the above strategies to determine whether a site is exploitable or not. In this regard, the above strategies assess various victim browser statuses. Each of them is performed on three different victim browser statuses:

(a) **Status A. No Cookie,** when the victim opens the attack URL, there is no cookie related to the targeted site in the victim's browser. In other

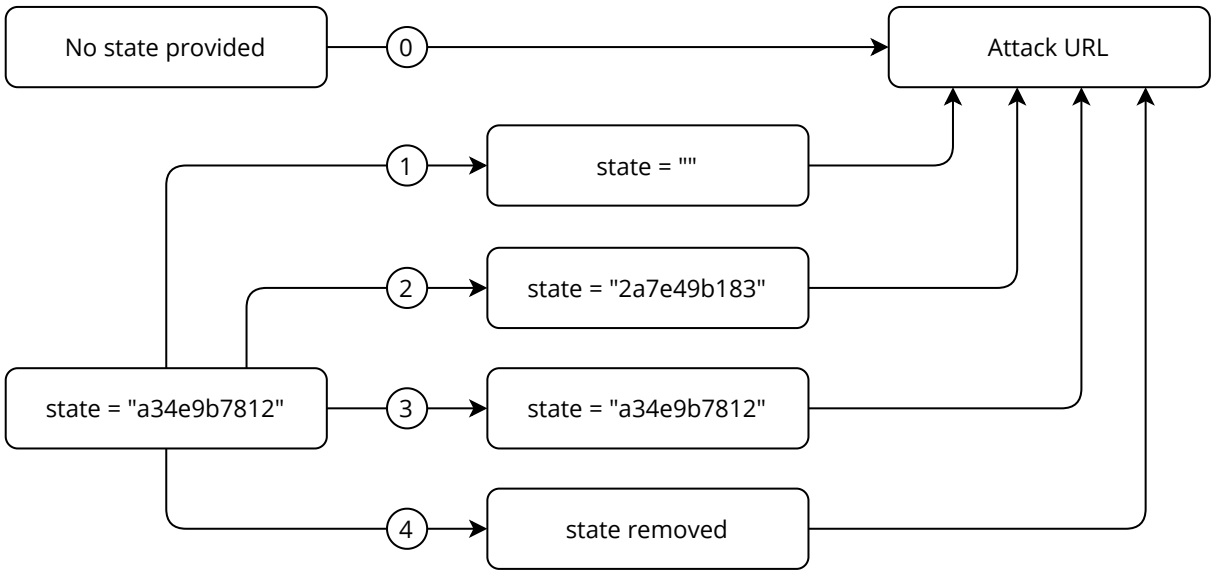

Figure 4.3: Abstract view of OCSRF detection methodology.

words, either victim never visited the targeted site in the past or used a new/history-cleared browser. Obviously, no cookies will be sent to the server when attack URLs are requested.

(b) **Status B. Visitor Cookies,** If the victim visited the targeted site in the past and visitor or unauthorized cookies have been set in the browser, the victim's browser adds them to all requests. In this case, the crawler visits the first page of the candidate site and stores all cookies before requesting the attack URLs.

(c) **Status C. Authorized Cookies,** If the victim is already authenticated to the targeted site, authorized cookies have been set on the victim's browser. To simulate this test case, the victim has been authenticated by our crawler by logging in to the victim's account before requesting the attack URLs.

Each of the created attack URLs, obtained from applying previously-mentioned strategies, would be tested on each and every browser status defined above. As we have five different attack strategies and three possible

victim browser statuses, we would end up with 15 test cases that would be exploited for each site. We later open all attack URLs inside victim browsers. We consider a test case to be successful if the attacker's marker is observed inside the victim's browser.

In a nutshell, each test case could be considered as the following three-step process.

1. Extract an attacker valid authorization response.
2. `state` parameter modification based on attack strategies.
3. Simulation of OCSRF attack on one of victim browser's status

4.3.4 Ethical Consideration

All test cases were performed with accounts specifically generated for this purpose. We never tried to exploit user accounts outside of our control. In all vulnerability assessment phases, our crawler never injected, sent, or stored any malicious payload to candidate websites. In order to evade detection by bots detector [113], less than 100 pages of each candidate site were visited slowly in the data collection phase. We also developed a Selenium crawler to complete the authentication steps and simulate a real user browser session. The number of requests involved in all test cases is significantly low, and the examined websites did not suffer from excessive bandwidth consumption. Moreover, all tests were conducted on the entire set of candidate websites; therefore, none of them has repeatedly been scanned in a short period of time.

Responsible Disclosure Since the impact of the discovered vulnerabilities is severe, we reported the site owners using recommended notification techniques[69, 105].

We contacted the vulnerable websites' owners through the support/security email or submitted a new request on their websites. We explained the

details of the OCSRF attack on their websites and how it could endanger the security of their users and provided some references regarding CSRF attacks. Some websites (about 9%) have replied asking for more details about the vulnerability and tried to fix the problem immediately. After one month of our report, we checked the rest (the ones that did not reply), and only a few websites fixed the vulnerability after our report.

Additionally, we tried to disclose the vulnerabilities to those websites for which a centralized reporting system such as Hackerone[56] can be used, as these promise an increased success rate over attempting direct notification.

4.4 Analysis

In this section, we present the results of the empirical analysis and discuss them in detail. We conducted a large-scale analysis implementing all the above attack scenarios to test OCSRF attacks in the wild. In this study, we discuss the measurement results of each attack strategy with different victim browser statuses. We managed to answer the following research questions:

(Q1) What is the popularity of OCSRF vulnerabilities on high-profile and popular websites?

(Q2) Can OCSRF vulnerabilities be exploited in the victim browser with unauthorized and authorized cookies?

(Q3) What are the most exploitable OCSRF attack strategies and test cases?

(Q4) Could the victim browser status be used to discover OCSRF vulnerabilities?

(Q5) What are the most popular implementation mistakes?

(Q6) How do the current mitigation mechanisms against OCSRF respond through the attack scenarios? Are they satisfactory enough to protect the victim?

4.4.1 Measurement Overview

Dataset We fed our crawler with the Alexa Top 50K websites and analyzed the first page of them to extract the list of candidate websites with OAuth login. Since we considered two types of SSO implementation, we targeted the most popular ones, examples of each, Google and Facebook, in web applications. The crawler discovered 624 websites with Facebook login, Google login, or both. In the next step, we tried to create two sets of accounts (victim and attacker) and recorded the successful OAuth flow on each site. We

narrowed down the dataset to 314 for Facebook and 284 for Google due to the exclusion of websites with incomplete account registration (e.g., Social Security Number, credit card, etc.) and unsuccessful account verification.

Alexa Ranking Our crawler analyzed all fifteen proposed test cases on the target dataset and discovered 114 out of 314 (%36.3) websites with Facebook IdP and 117 out of 284 (%41.2) websites with Google IdP to be exploitable by at least one attack scenario. Given the distribution of the targeted and vulnerable websites across the Alexa Top 50K, it is noteworthy that about 32% of the websites among the Top Alexa 1K are vulnerable. Websites with higher Alexa ranking are slightly more vulnerable, but no specific major correlation among different buckets has been observed.

Login flow extraction This stage's goal is to gather the data the crawler needs to carry out the login automatically. The crawler must be able to successfully navigate from the target website's login page to the identity provider's authorization endpoint in order to complete this task.

In general, various websites do not consistently integrate SSO login with the same external identity provider. In some instances, the login button has a direct link to the endpoint of the authorization server.

Using an internal page with a redirect to the IdP login is an alternate method. Delegating the redirect to a JavaScript script is another method of getting to the login form. The realization of a standard technique to automate the login is complicated by the varied implementations of the login functionality. To correctly manage the widest range of potential scenarios, we devised an algorithm taking these differences into account.

Direct link Some websites create a direct link to the external IdP's authorization endpoint and put it in the corresponding login button. In this case, parsing the page of the evaluated website that contains the button is sufficient to extract the whole login URL.

Scenario	Example
1. Direct link	`<a href="https://www.idp.com/login..."></a>`
2. Page-generated redirect	`<a href="https://example.com/login?type=fb"></a>`
3. JavaScript-generated redirect	`<a id="idp-login" onclick="login()"></a>`

Table 4.1: Login scenarios handled by the crawler

The crawler simply requires the address of this page in order to start the login procedure. A regular expression that identifies potential addresses connected to the IdP login endpoint defines the search pattern. The following regex is used to find the link to the login page:

```
https://www\.idp\.com/((v\d*\.?\d*/)?dialog/oauth|login\.php|login/reauth\.php).*
```

As specified in the regular expression, the base URL `https://www.idp.com` is optionally followed by a path segment containing a floating-point number. This component designates a certain login dialog version. According to the documentation, it is not required to include the version number; unversioned calls always lead to the API's earliest available implementation.

Page-generated redirect If the investigated website contains an internal page that redirects to the identity provider's authorization endpoint, the crawler can save the URL and utilize it later to find the external login page. Because it does not require additional investigation, the presence of a page-generated redirect is frequent and represents the simplest scenario to handle.

To perform the login, the crawler just has to access the website's internal page URL to log in; the content produces the external login URL and directs the browser to the IdP's authorization endpoint.

With the aim of finding internal pages redirecting to the IdP, the Beau-

tifulSoup [92] parsing library is used for the analysis of the login page. The search process starts by choosing all HTML elements with the `href` property. A resource's location is specified by the hypertext reference, which also establishes the connection between the source element and the destination anchor. Absolute and relative URIs pointing to the external identity provider's authorization endpoint are of our interest.

The page source is used to extract any links that include a keyword associated with the identity provider. To convert relative URL paths to their absolute paths and create a list of addresses the crawler could test, the `urljoin` method of the `urllib.parse` package was utilized.

JavaScript-generated redirect The information about the button's location is saved for the website if the identity provider redirection is started by JavaScript and triggered by a button click event. The sign-in buttons were found using a keyword-based technique; these elements typically have some keywords identifying the login action and the identity provider.

The login button was identified, and an XPath expression [41] encoding the element's position was stored in a file-based database using a parsing library to analyze the HTML source of each website's login page. The crawler uses this expression to find the element in the HTML source needed for the login procedure. The crawler is equipped to press the button, cause the redirection, and then arrive at the IdP login page.

Verification Since the login flow extraction is done automatically, the crawler attempting to access each website's identity provider's login page verifies the accuracy of the data collected. If the process is successful, the acquired data is designated as verified; if not, some manual analysis is done to fix the stored data.

Cookie-based login Every web application needs to be independently

tested in a fresh browser instance. At the end of each test case, the browser must be closed and reopened in order to conduct each test in a distinct environment.

We did not use the form-based login during the test phase to prevent a sizable and possibly suspicious number of consecutive logins. This is an added safety measure to avoid having a large number of automatic logins in a short period of time recorded by the identity provider.

For each identity provider, we made two accounts during the registration process. All of the study's participating websites use these profiles to authenticate users. Additionally, each web application's registration process had to be manually completed in order to create client-side accounts. Disabling an IdP account as a result of a ban would have required repeating the relevant sign-up procedures. For this reason, we thought of various approaches that might be used to lessen the likelihood of a ban.

As already indicated, the login form is not required for user authentication. Cookies can be imported and exported using Selenium. Once the form-based login on the authorization server was completed, the session cookies that were produced were kept for further use. The imported saved cookies restored the login status when the browser was launched. Using this method, we were able to confirm that the login procedure was accurate for each website and for all attack scenarios that were put to the test.

4.4.2 Results

Categories based on presence of `state` The candidate websites have been categorized based on the absence or presence of `state` parameter within the recorded authorization request. For the former category, as mentioned in 4.3.3, our crawler directly exploits the site without `state`, and no modification is applied to the attack URLs. However, in the latter category, due to the presence of state parameters, 15 different attack scenarios have been

tested. We will discuss the result of both categories in 4.4.2 in detail. The overall results of the crawler are summarized as follows:

1. The first category, 44 out of 314 (14.0%) websites with Facebook IdP and 65 out of 284 (22.9%) websites with Google IdP do not use `state`, which shows a significant increase in utilization of `state` compare to past large scale analyses[71, 64, 30]. It is worth mentioning that absence of `state` does not guarantee successful exploitation of OCSRF as our crawler found 41 out of 44 (93.1%) with Facebook IdP and 60 out of 65 (92.3%) with Google IdP are exploitable; we will discuss case studies in 4.4.2.

2. The second category, 270 out of 314 (85.9%) websites with Facebook IdP and 219 out of 284 (77.1%) websites with Google IdP are using `state`. Although this indicates a significant increase in OCSRF protection compared to the past studies[71, 64, 30, 72], our crawler detected 73 out of 270 (27.0%) with Facebook IdP and 57 out of 219 (26%) with Google IdP exploitable websites utilizing different test cases. This high number indicates the complexity of OCSRF protection implementation. We will discuss the case studies in details in section 4.4.3.

Table 4.2: Number of exploitable sites in Facebook by OCSRF for each attack scenario

Attack	**No cookies (a)**	**Visitor cookies (b)**	**Auth. cookies (c)**	**All**
0	33 (10.5%)	41 (13.1%)	23 (7.3%)	41 (13.1%)
1	34 (10.8%)	33 (10.5%)	23 (7.3%)	41 (13.1%)
2	30 (9.6%)	40 (12.7%)	23 (7.3%)	40 (12.7%)
3	49 (15.6%)	63 (20.1%)	36 (11.5%)	64 (20.4%)
4	33 (10.5%)	34 (10.8%)	24 (7.6%)	40 (12.7%)
Total	91 (29.0%)	105 (33.4%)	62 (19.7%)	114 (36.3%)

Table 4.3: Number of exploitable sites in Google by OCSRF for each attack scenario

Attack	No cookies (a)	Visitor cookies (b)	Auth. cookies (c)	All
0	40 (14%)	60 (21.1%)	24 (8.4%)	60 (21.1%)
1	29 (10.2%)	32 (11.3%)	13 (4.5%)	37 (13%)
2	26 (9.1%)	31 (10.9%)	14 (4.9%)	33 (11.6%)
3	35 (12.3%)	48 (17%)	25 (8.8%)	50 (17.6%)
4	29 (10.2%)	32 (11.3%)	13 (4.6%)	36 (12.7%)
Total	80 (28.2%)	111 (39%)	50 (17.6%)	117 (41.2%)

Attack Strategies Table 4.2 for Facebook IdP and Table 4.3 for Google IdP shows the number of exploitable websites to each attack strategy. As shown, the «attack strategy 3: Unlinked `state`» has the highest success rate(20.4% for Facebook and 17.6% for Google) in all victim browser statuses. In this attack strategy, as previously described in 4.3.3, the victim visited a crafted attack URL with an attacker's valid and unused `state`. It means lack of proper relationship between the victim browser and generated `state` is the most common implementation mistake. Interestingly «attack strategy 1: Empty state» has the second rank in both SSO types, which means some websites mistakenly accept the authorization link with an empty `state` value.

Attack Scenarios Since visitor cookie is the most vulnerable status, which makes the highest success rate (20.4% with Facebook IdP and 17.6% with Google IdP) and «attack strategy 3: Unlinked `state`» is the most effective attack strategy, attack scenario «3b» has the highest detection rate. Our crawler detected 63 out of 270 (20.1%) websites with Facebook IdP and 48 out of 219 (21.9%) with Google IdP to be exploitable with it. Attack scenarios «1c» and «2c» had the lowest detection rates, most probably because targeted websites do not accept new OAuth login when a user is authenticated.

Victim Browser Status We tested each attack strategy with three dif-

Table 4.4: Classification of exploitable sites in Facebook by OCSRF - The first category of candidates (with Absence of `state` parameter)

#	0a	0b	0c	Sites
1	●	●	●	17 (38.6%)
2	●	●	○	16 (36.4%)
3	○	●	●	6 (13.6%)
4	○	○	○	3 (6.8%)
5	○	●	○	2 (4.5%)
Total	33	41	23	44

Table 4.5: Classification of exploitable sites in Google by OCSRF - The first category of candidates (with Absence of `state` parameter)

#	0a	0b	0c	Sites
1	●	●	●	17 (26.1%)
2	●	●	○	23 (35.3)
3	○	●	○	13 (20%)
4	○	●	●	7 (10.7%)
5	○	○	○	5 (7.6%)
Total	40	60	24	65

ferent victim browser status. Our crawler detects unique exploitable cases in each browser status. Previous researches only test the OCSRF in a clean browser without the presence of any cookie[107] or only with visitor cookie[125]. In this research, our crawler was able to detect 23 out of 114 (20.2%) with Facebook IdP and 37 out of 117 (31.6%) with Google IdP more exploitable OCSRF cases compared to test case «a: No cookies» through utilizing different browser status and 9 out of 114 (7.9%) with Facebook IdP and 6 out of 117 (5.1%) with Google IdP compared to test case «b: Visitor cookies». Applying all of the browser statuses together with attack strategies has been done for the first time to the best of our knowledge.

Based on our results presented in Table 4.2 for Facebook and Table 4.3 for Google, the presence of visitor cookies in the victim browser increases the chance of finding exploitable cases significantly. Even though it is common that websites with authorization cookies are less vulnerable, we observed

Table 4.6: Classification of exploitable sites in Facebook by OCSRF - The second category of candidates (with Presence of `state` parameter)

#	1a	1b	1c	2a	2b	2c	3a	3b	3c	4a	4b	4c	Sites
1	○	○	○	○	○	○	○	○	○	○	○	○	197 (73.0%)
2	●	●	●	●	●	●	●	●	●	●	●	●	18 (6.7%)
3	○	○	○	○	○	○	●	●	●	○	○	○	11 (4.1%)
4	●	●	○	●	●	○	●	●	○	●	●	○	7 (2.6%)
5	○	○	○	○	○	○	●	●	○	○	○	○	6 (2.2%)
6	○	●	○	○	●	○	○	●	○	○	●	○	5 (1.9%)
7	○	○	○	○	○	○	○	●	○	○	○	○	4 (1.5%)
8	●	○	○	○	○	○	○	○	○	●	○	○	4 (1.5%)
9	○	○	○	●	●	●	●	●	●	○	○	○	3 (1.1%)
10	○	○	○	○	●	○	○	●	○	○	○	○	3 (1.1%)
11	●	○	○	○	○	○	○	○	○	○	○	○	2 (0.7%)
12	●	○	●	○	○	○	○	○	○	●	○	●	2 (0.7%)
13	○	○	○	●	●	○	●	●	○	○	○	○	2 (0.7%)
14	○	●	●	○	●	●	○	●	●	○	●	●	2 (0.7%)
15	○	○	○	○	○	○	●	●	●	●	●	●	1 (0.4%)
16	○	○	○	○	○	○	●	○	○	○	○	○	1 (0.4%)
17	●	●	●	○	○	○	○	○	○	●	●	●	1 (0.4%)
18	○	○	○	○	○	○	○	●	●	○	○	○	1 (0.4%)
Total	34	33	23	30	40	23	49	63	36	33	34	24	270

websites that unexpectedly were vulnerable only in these specific test cases, which would be discussed in 4.4.2 and 4.4.2.

`state` Parameter

As mentioned, there are two categories of candidates based on the presence of `state` parameter within the recorded authorization request, which will be analyzed and explained separately in this section.

Absence of `state`

Interestingly, 44 out of 314 (14.0%) of websites on Facebook and 65 out of 284 (22.9%) of websites on Google do not set `state`. In some cases, the absence of visitor cookies led to errors in the OAuth login flow, and this contributes to explaining the lower number of vulnerabilities found in status «a» than «b».

All exploitable websites are also exploitable to «b» while about half of

them are not exploitable when there is an authorized cookie. The classification of exploitable websites are listed in table 4.4 and table 4.5. Each row represents one pattern w.r.t different test cases (1a,1b,etc.). A filled circle in each entry indicates successful exploitation. The `Sites` column shows the total number of websites that have been found exploitable via the indicated pattern in the corresponding row. Take Facebook, for example; 3 out of 44 websites were not exploitable to any of the attack scenarios, and so on. While only 17 websites are vulnerable to all three attack scenarios, there are two websites that are only exploitable when the visitor cookies are present. It means successful exploitation of them requires the victim browser to add only unauthorized cookies in the Attack URL.

In contrast to other research, the absence of `state` does not guarantee successful exploitation of OCSRF, as other enabling factors can prevent targets from being exploited. In order to remove the false positives, our crawler analyzed all the websites in the first category of candidates without `state` parameter. Unexpectedly, three websites with Facebook IdP and five websites with Google IdP were not exploitable. In this regard, some websites use encoded and nonstandard parameters in the `redirect_uri` and implement proper validation to check if the OAuth flow is initiated with the same browser. At the time of writing this paper, OAuth specifications for both Facebook and Google do not allow developers to set arbitrary parameters to `redirect_uri` as the full redirect URL should be reserved, and the OAuth flow is blocked if there is any change in `redirect_uri`.

However, other websites expect the flow to be completed in a popup window, which is not opened by the crawler during the attack execution. The JavaScript code running on the client-side fails due to the absence of an opener parent window, and the attack is consequently blocked in the browser. We consider this site a secure one despite the absence of adequate protection against OCSRF. We will discuss related case studies in 4.4.3.

Table 4.7: Classification of exploitable sites in Google by OCSRF - The second category of candidates (with Presence of `state` parameter)

#	1a	1b	1c	2a	2b	2c	3a	3b	3c	4a	4b	4c	Sites
1	○	○	○	○	○	○	○	○	○	○	○	○	164 (74.8%)
2	●	●	●	●	●	●	●	●	●	●	●	●	10 (4.6%)
3	○	○	○	○	○	○	●	●	●	○	○	○	7 (3.2%)
4	●	●	○	●	●	○	●	●	○	●	●	○	12 (5.5%)
5	○	○	○	○	○	○	○	●	○	○	○	○	3 (1.4%)
6	○	○	○	○	○	○	○	●	●	○	○	○	4 (1.8%)
7	○	●	○	○	●	○	○	●	○	○	●	○	4 (1.8%)
8	○	○	○	●	●	●	●	●	●	○	○	○	2 (0.9%)
9	●	○	○	●	○	○	●	○	○	●	○	○	2 (0.9%)
10	○	●	○	○	○	○	○	●	○	○	●	○	1 (0.5%)
11	○	●	○	○	○	○	○	○	○	○	●	○	1 (0.5%)
12	●	○	○	○	○	○	○	○	○	●	○	○	2 (0.9%)
13	●	●	○	○	○	○	○	○	○	●	●	○	1 (0.5%)
14	○	○	○	○	●	○	○	●	○	○	○	○	1 (0.5%)
15	○	●	●	○	●	●	○	●	●	○	●	●	2 (0.6%)
16	●	○	●	○	○	○	○	○	○	●	○	●	1 (0.5%)
17	○	○	○	○	○	○	●	●	○	○	○	○	1 (0.5%)
18	●	●	○	○	○	○	●	●	○	●	●	○	1 (0.5%)
Total	29	32	13	26	31	14	35	48	25	29	32	13	219

Presence of `state` Attack

The presence of the `state` does not mitigate OCSRF vulnerabilities. We summarized each exploitable pattern that was observed during our experiment on websites in our candidate set in Table 4.6 for Facebook and Table 4.7 for Google. About 73% of websites are not exploitable to any of the proposed attack scenarios. In many websites, this is due to a correct implementation of the OAuth flow. Some secure instances notify the user about the OCSRF attack; others simply display a generic authorization error or do not perform any action. It should be noted that the group of 197 websites on Facebook and 164 websites on Google marked as not exploitable by the crawler may contain a small fraction of false negatives. This hypothesis is supported by some evidence presented later in the analysis. For this reason, the number of vulnerabilities identified in our tests must be considered as a lower bound. Details will be discussed in the following section.

Custom HTTP Headers

In our database, 159 websites out of 395 (40%) implemented their login actions via XMLHttpRequest containing the X-Requested-With header. As mentioned, setting this custom header to requests defends against the Login CSRF attack; however, we found that 50 websites out of 159 websites (31.4%) were still vulnerable.

Surprisingly, the websites did not set this custom header in all attack scenarios and had different behavior towards each attack. As we divided the candidates into two categories (no state and with the state), out of 159 websites, 23 websites set no state parameter (first category), and 136 websites set the state parameter (second category). Out of 15 proposed attack scenarios, three ones applies to the first category, and the other 12 apply to the second category. With all consideration, if 159 websites implement this defense in all attack scenarios, there should be 1,701 occurrences. Instead, there were 1,063 occurrences in total. All 109 non-vulnerable websites behaved consistently in all attack scenarios and implemented this defense correctly, and the custom header was present in all test cases. Presenting occurrences for each victim's browser status are as follows: 254 for no cookie (a), 280 for visitor cookies (b), and 529 for authorized cookies(c). The custom header is mostly set when the victim is already logged in and the authorized cookies are present.

On the other hand, 50 vulnerable websites were inconsistent in each attack scenario, and they did not include the custom HTTP header in every test case. Table 4.8 shows the number of vulnerable websites for each test case, representing a great difference in various victim browser statuses. Most of them (42 out of 50 websites) are still vulnerable to visitor cookies status (b), and the least number (27 out of 50 websites) belongs to authorized cookies status. Comparing the attack strategies, the third one (Unlinked state) is

Table 4.8: Number of exploitable sites with custom header for each attack scenario

Attack	**No cookies (a)**	**Visitor cookies (b)**	**Auth. cookies (c)**	**All**
0	12 (7.5%)	19 (12%)	13 (8.2%)	20 (12.6%)
1	10 (6.3%)	10 (6.3%)	7 (4.4%)	15 (3.1%)
2	7 (4.4%)	10 (6.3%)	5 (3.1%)	10 (6.3%)
3	17 (10.7%)	22 (13.9%)	10 (6.3%)	24 (15.1%)
4	10 (6.3%)	9 (5.7%)	9 (5.7%)	16 (10%)
Total	33 (20.7%)	42 (26.4%)	27 (17%)	50 (31.4%)

the most successful one among the attack strategies. However, 12 out of 50 websites implemented the custom header in all the test cases and were still vulnerable in all situations. It is worth mentioning that 7 out of 12 websites belonging to the first category (no state).

In some cases, the custom header was included after the attack was taken successfully (in case of (a) and (b) victim browser status). As shown in classification 4,13,18 in Table 4.7, classification 4,13 in Table 4.6 and classification 2 in both Table 4.5 and 4.4, the websites are more vulnerable in (a) and (b) victim browser status. In the case of authorized cookies browser status (c), the custom header set in both situations, before and after the attack is taken, and yet this mitigation was not effective against OCSRF, as shown in classification 2,3,8,6,15 in 4.7, in classification 2,3,9,14 in 4.6, in classification 1,3 in 4.4 and in classification 1,4 in 4.5.

Generally, the custom HTTP header is more implemented and also more effective when the user is already logged in (authorized cookies status) compared to other victim browser statuses. So, it is one of the reasons the number of vulnerable websites in authorized cookies status is much lower than in the other two victim browser status.

User Interaction

This mitigation performs through some techniques (e.g., re-authentication, CAPTCHA, or One-time token). Based on our observation in the dataset, only 25 out of 395 websites (6%) utilized this mitigation to prevent CSRF attacks; Only one site performed CAPTCHA, and others (24 websites) went through a re-authentication technique for this type of defense.

Any site without OAuth service, utilizing a successful re-authentication technique against CSRF mitigation, redirects the user to the main login page of the site. However, within OAuth protocol, when the re-authentication technique occurs, the user will be redirected to the IdP(Google/Facebook) login page, requiring the Google/Facebook credentials.

In our experiment, proposing 15 different attack scenarios, this technique occurred in case of (a) and (b) victim browser status and none in case of (c) authenticated cookies. This technique was successfully performed in the victim browser (in the case of C victim browser status), and the reason it did not require the re-authentication by the user is that the authenticated cookies are already present in the victim browser, and as a result, the login CSRF attack in this case (c) was not successful, redirecting the victim to his own account instead of attacker's account.

As shown in other mitigation against OCSRF, the mitigation provided in the websites are not consistent in all the attack scenarios, and some factors (e.g., the victim browser status and fuzzing the state parameter value) change the course of CSRF mitigation, leading to a successful attack. This mitigation is no exception: While 19 websites performed this mitigation in all attack scenarios stopping the OCSRF attack successfully, six websites were inconsistent with utilizing this mitigation, causing the websites to be vulnerable in some scenarios.

4.4.3 Discussion

Case Studies

In this section, different attack scenarios used during this research will be explained, along with notable case studies of each attack strategy. It is worth mentioning that in this section, the second category, the presence of `state` parameter, is studied.

Empty & Missing `state` In attack strategy 1, the value of the `state` in the authorization response is replaced with an empty string. At the beginning of the flow, the site generates a valid `state` to identify the authorization request. If the authorization response contains an empty `state` value, the application is supposed to not accept it and block the OAuth flow. The same approach applies to attack strategy 4, in which the `state` parameter – not only its value – is entirely removed from the authorization response URL.

A couple of manually analyzed websites have been discovered to be exploitable only to attacks 1 and 4, as illustrated in Table 4.6, classification 8, 12, 17 for Facebook, and in Table 4.7, classification 11, 12, 13, 16 for Google. This result shows that when the parameter is present, `state` is handled supposedly and would be verified by the application. However, when the `state` value is empty, or the parameter is missing, the validation is bypassed, and the flow is successfully accepted. The application source code is not directly available. However, we can get an insight into the internal logic of the `state` validation algorithm by analyzing the site reactions in response to different inputs. This behavior can be clarified by following programming exemplification:

```
If state in response and state:
    if not is_valid(state):
        # Raise an exception and block the flow
```

```
# Continue and complete the flow
```

In Facebook SSO implementation, a couple of websites are exploitable only via attack strategy 1 but not the 4th (Refer to 4.6, classification 11). The validation process checks the presence of a parameter called `"state"` in the authorization response and blocks the flow if it is not found. However, an empty `state` is accepted as valid and leads to the flow completion. The reverse is still possible when a site is exploitable with attack 4 and not to 1 (Refer to Table 4.6, classification 15). As an instance, we found a case study in which the verification succeeds only in the presence of a valid `state`, while it could be bypassed if the parameter was not provided. The empty `state` supplied in the first test scenario was considered invalid by the application and caused the flow to be halted.

Furthermore, we also found six exploitable websites in which the only performed validation is related to the presence of the `state` parameter inside the authorization response (For Facebook in Table 4.6, classification 9 and 10. For Google in Table 4.7, classification 8 and 14). The client application does not accept requests with a missing or empty `state` parameter, but even a random value is enough to bypass the validation.

The difference between attack strategies 1 and 4 is subtle, and the results are almost overlapping. But the insight provided by the above-mentioned unexpected results would be to take both attack strategies into account to discover related vulnerabilities to a great extent.

Unlinked `state` In attack strategy 3, the authorization response received by the attacker is maintained unchanged and sent to the victim. The test is performed to assess the absence of a valid relationship between the `state` and the user's session. If the `state` is not handled properly during the generation of the authorization request, the application does not have enough information to perform correct validation in the subsequent steps of the flow.

The site is not able to understand whether the authorization response was

issued by the identity provider for the current user or if someone else initiated the request. As a result, the client may accept all the `state` values produced by the application as valid.

As illustrated in Table 4.2 for Facebook and 4.3 for Google, attack strategy 3 is the most successful one. More than 20% of the candidate site are vulnerable to scenario «3a», «3b», or «3c» for both Google and Facebook in total. This can be justified by the inherent complexity of implementing a valid relationship between the browser session and the `state`, which requires generating and storing a random token and proper management of that in the validation phase.

Even though the RFC clearly describes the role and operation of the `state` parameter, the documentation provided by different identity providers is not often sufficiently precise and detailed. For Facebook, 23 out of 114 (20.2%) and for Google, 15 out of 117 (12.8%) exploitable websites are only vulnerable to attack strategy 3 (In Table 4.6, classifications 3,5,7,16, and 18. In Table 4.7, classification 3, 5, 6 and 17). For these applications, arbitrary `state` values are correctly rejected by the validation method, but valid states with incorrect associated with user sessions are erroneously not refused.

Eleven websites with Facebook IdP and seven websites with Google IdP are vulnerable to all configurations of attack strategy 3 (Table 4.6, classifications 3 and Table 4.7, classifications 3). In total, five websites were incorrectly classified by the crawler in this group due to the notification message to the victim, which we considered as false positives. The sample message is shown as follows.

> «The account ({attacker email}) belongs to another user ({attacker name}). You can log in as {attacker name} or cancel to stay logged in as {victim name}.»

The crawler wrongly classified websites due to the presence of marker

information related to the attacker's account. The attack is not blocked by default, but the above-mentioned informative message helps the victim to make the correct decision. Although not recommended, we classified the websites which show their users a similar message as secure (not vulnerable).

Lack of `state` Validation In attack strategy 2, the `state` parameter produced by the client application is replaced with another string which is a random permutation of the initially generated value. The new parameter has the same length as the original and the same character set. The purpose of this strategy is to understand if using an invalid `state` is sufficient to bypass the OCSRF protections implemented by the examined websites.

For Facebook, our crawler detects total number of 30 out of 114 (26.3%), 40 out of 114 (35.1%) and 23 out of 114 (20.2%) exploitable websites to be vulnerable to test cases «2a»,«2b», and «2c».

For Google, the numbers are 26 out of 117 (22.2%), 31 out of 117 (26.5%), and 14 out of 117 (12%), respectively. The presence of visitor cookies increases the attack success rate, similar to other attack strategies.

It can be easily noticed from Table 4.6 and Table 4.7 that the results of attack strategies 2 and 3 are strongly related.

There are no websites discovered to be vulnerable only to 2, and the websites vulnerable to this attack constitute a proper subset of the ones vulnerable to the third scenario. Although it does not add any item to the set of vulnerable websites, the second scenario gives remarkable indications about the nature of the validation performed. For instance, looking at the attack results reveals the possibility of a completely incorrect validation from a session association issue.

Specific Characters in generating `state`: Websites that employ a `state` consisting of a single character or a string of N identical characters are an example of a specific situation for attack strategy 2.

With this approach, it is impossible to generate a distinct permutation of

the original string, and the attack cannot be performed as it was intended.

There are two websites that have `state` of length, one among the samples taken into account. The symbols are respectively underscored «_» and slash «/». The website utilizing «/» was discovered to be open to attack scenario 3b. Even if «/» is replaced with another random character, the attack still succeeded. The attack failed on the other site because the random character was used in place of «_», blocking its execution.

Based on the aforementioned implementation errors, we advise using a `state` value that is generated randomly and cannot be guessed at. Furthermore, in order to prevent OCSRF vulnerability, `state` must be correctly associated with the user session.

Popup-based login

Some websites' login screens include popup windows. When carrying out attacks, the developed crawler handles opening several browser windows and transitioning control from one to the other with accuracy. Selenium is equipped to detect whether a secondary window is open or closed and to respond appropriately.

Utilizing a popup can occasionally offer some degree of defense against attacks. The OCSRF cannot always be completed by the crawler. For instance, one of the examined domains has a constant `state` option, but all tests show that it is not vulnerable.

A popup window appears when the «Login with Facebook/Google» button is clicked. The designed browser extension correctly collects the attacker-generated authorization response. The victim's browser session opens the URL, leading to the target website's redirection endpoint.

A function of the `window.opener` object is called in a few lines of JavaScript that are part of the page response. Since there is no `opener` window and the URL is called directly from the victim's browser address bar, an error is

issued, and the attack fails.

We discovered another domain that is safe for the same reason. Because the login flow involves communication between a parent window and a popup, the OAuth flow cannot be completed in the victim's browser. The attack cannot be carried out because the victim's browser does not contain this structure. The attack can be carried out successfully if the authorization response is manually accessed from the login prompt.

The fact that this login structure does not stop the attack from being successful via additional methods means that it cannot be used as an efficient OCSRF mitigation. For instance, a POST request that is carefully prepared to launch an attack against the domain with the constant `state` parameter results in vulnerability. The authorization code is given as a body parameter in a script that produces a POST request to an internal endpoint when the login popup calls the JavaScript function in the main window.

In order to obtain a valid access token, the client then requests the authorization server to complete the flow. Not even the `state` parameter is taken into account. It is sufficient to duplicate the POST request using a form, from a domain controlled by the attacker, in order to bypass the error and complete the attack.

Popup-based logins do not always stop a crawler from performing an attack. We discovered numerous websites using this access approach, many of which had undergone thorough testing and analysis.

Forcing the enabling factors

As shown, the attack scenarios mostly fail whether the victim is authenticated («c» configuration) and are most successful with «a» and «b» configurations.

By directing the victim's browser to the logout endpoint in the absence of security, the attack can be carried out with ease. A victim could be tricked into performing an unwanted logout action. As a result, it enforces the

authenticated user into a condition of «b» (visitor cookies), and the success rate of the OCSRF attack increases.

The usage of a logout form adequately protected with an anti-forgery token helps reduce the risk of this type of attack. The request's token is compared to the token saved in the browser session by the server. The logout action is performed if the validation is successful and the request is considered legitimate; otherwise, the activity is prevented. With the help of this protection, a logout request made from an attacker-controlled environment cannot be completed.

Constant `state`

The OAuth 2.0 specification clearly states that the `state` parameter must be one-time use and a random string. This requirement is necessary to protect applications from brute-force attacks. Some websites do not follow these instructions and include a fixed and constant `state` in the OAuth authorization request, which would not change for different users and browser sessions. These websites are not able to distinguish between a legitimate authorization response created for the victim and a response forged by the attacker.

We visited all candidate websites twice from two different browser sessions and compared the `state` values in order to identify this implementation problem. If the `state` remains unchanged, the site is potentially vulnerable to a "state reuse" attack. Our crawler collected and stored all authorization requests. We later extracted the `state` values from the URL and compared them to each other. The analysis disclosed 17 out of 270 (6.3%) websites with Facebook IdP and 17 out of 219 (7.7%) websites with Google IdP reusing the same `state` values.

Table 4.10 and 4.9 in the last column shows the number of discovered vulnerable websites with constant `state` parameter for each classification for Facebook and Google respectively. 16 out of 17 (94.1%) on Facebook

Table 4.9: Classification of exploitable sites in Google by OCSRF - Constant `state` parameter

#	1a	1b	1c	2a	2b	2c	3a	3b	3c	4a	4b	4c	Sites
1	○	○	○	○	○	○	●	●	●	○	○	○	2 (11.8%)
2	●	●	●	●	●	●	●	●	●	●	●	●	5 (29.4%)
3	○	○	○	○	○	○	○	●	○	○	○	○	2 (11.8%)
4	○	○	○	○	○	○	○	●	●	○	○	○	2 (11.8%)
5	○	●	○	○	●	○	○	●	○	○	●	○	1 (5.9%)
6	○	○	○	●	●	●	●	●	●	○	○	○	1 (5.9%)
7	●	●	○	●	●	○	●	●	○	●	●	○	1 (5.9%)
8	○	○	○	○	○	○	○	○	○	○	○	○	2 (11.8%)
9	●	●	○	○	○	○	●	●	○	●	●	○	1 (5.9%)
Total	7	8	5	7	8	6	10	15	10	7	8	5	17

Table 4.10: Classification of exploitable sites in Facebook by OCSRF - Constant `state` parameter

#	1a	1b	1c	2a	2b	2c	3a	3b	3c	4a	4b	4c	Sites
1	○	○	○	○	○	○	●	●	●	○	○	○	5 (29.4%)
2	●	●	●	●	●	●	●	●	●	●	●	●	4 (23.5%)
3	○	○	○	○	○	○	○	●	○	○	○	○	3 (17.6%)
4	○	●	●	○	●	●	○	●	●	○	●	●	1 (5.9%)
5	○	●	○	○	●	○	○	●	○	○	●	○	1 (5.9%)
6	○	○	○	○	○	○	●	●	●	●	●	●	1 (5.9%)
7	○	○	○	○	○	○	○	○	○	○	○	○	1 (5.9%)
8	○	○	○	○	○	○	●	●	○	○	○	○	1 (5.9%)
Total	4	6	5	4	6	5	11	16	11	5	7	6	17

and 15 out of 17 on Google were found vulnerable to the CSRF against `redirect-uri`. A manual analysis showed that the use of a popup-based login prevented the completion of OCSRF attack.

The presence of a constant `state` value does not provide any additional protection to the OAuth flow as a malicious user can easily assess the existence of a "state reuse" vulnerability and include the same unchanged parameter in every attack attempt. Finally, a couple of websites are classified as not vulnerable by the crawler. (Table 4.10, classification 7 and Table 4.9, classification 8).

State parameter value

In the case of OCSRF attacks, the `state` parameter plays a crucial role. This is why we retrieved and looked at the values that the tested websites set. The distribution of the `state` parameters utilized by the examined applications, by length, is shown in Figure 4.4 length. The number of states with lengths less than or equal to the value indicated on the x-axis is represented by each column of the histogram.

Out of the 395 tested websites, 109 do not include a `state` parameter. The `state` values contain characters in the range [32, 64] in more than 71% of the cases. There are 11 websites with a `state` length equal to or less than 5.

The length of the Nonce used to apply for the OCSRF protection frequently does not match the length of the parameter. When the `state` value is used to convey extra data like paths, URLs, or application-specific information, this happens. The information is often organized in a JSON structure by these enriched states, and the CSRF token is added as an object attribute.

A quick manual check found some websites that lacked a random string of characters in the `state`. The parameter is absolutely ineffective for defending the website from OCSRF attacks since it lacks a random value with enough

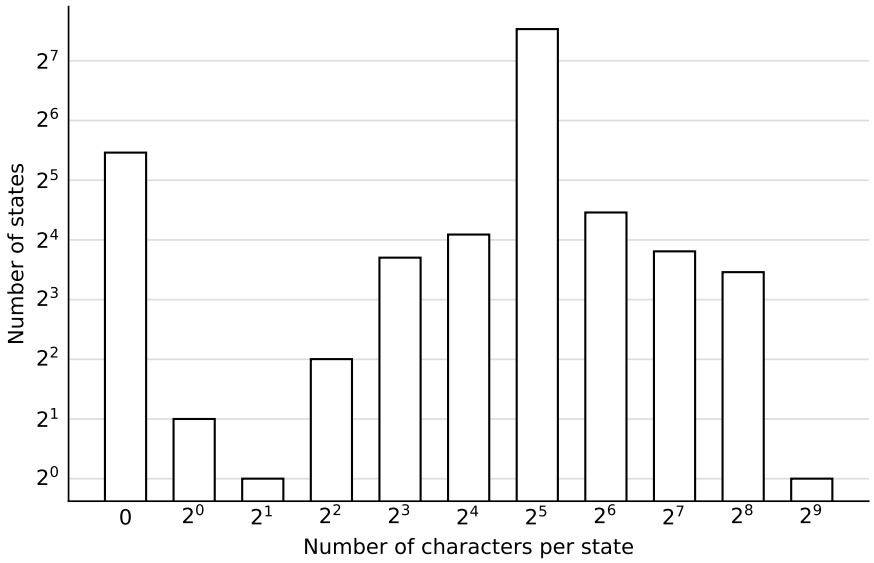

Figure 4.4: Classification of login `state` parameter length

entropy.

The absence of an OCSRF defense is clearly seen when the `state` value is readable. For instance, some websites include their base URL in the `state`. Even if the information relating to the `state` is concealed behind a general encoding technique, the misuse of this security parameter can be easily detected. `State` parameters with a Base64 encoding are present on a number of websites. Encoding is frequently employed in this context to serialize JSON values.

Other CSRF defense mechanisms

As an example, we discovered one application that is not exploitable, using the SameSite cookie attribute in addition to the `state` parameter [86] against Login CSRF. SameSite is a cookie attribute that instructs the browser on whether to send cookies along with cross-site requests.

The application opens a popup window during the login process. When the login popup calls the JavaScript function in the main window, a script

generates a POST request to an internal endpoint, providing the authorization code as a body parameter. The client subsequently continues the flow, contacting the authorization server to receive a valid access token.

This cookie attribute can take the values Lax, Strict, or None. In the example above, the application has set the SameSite to Lax. The Lax value will be sent only in the GET request in top-level window navigations and blocks the cross-site requests in CSRF-prone request methods such as POST.

It is worth mentioning that this attribute should not be replaced with a CSRF Token and only implemented as an additional layer of defense. This attribute alone protects the user through the browsers supporting it, and it could be bypassed easily by replicating the POST request using a form from a website that is under the attacker's control.

In double submit cookie [87], a site can set a CSRF token as a cookie and also insert it as a hidden field in each HTML form. When the form is submitted, the site can check that the cookie token matches the form token. This mechanism is stateless since it does not require any server-side state for CSRF token.

In table 4.6 and table 4.7, classification 4, the websites are vulnerable in all configuration, except the authorized cookies browser status (c). We inspected the HTTP traffic and discovered seven websites with presence of CSRF token in cookies, as '`xf_csrf`', and an HTTP header, as '`X-CSRF-TOKEN` or '`X-XSRF-TOKEN`'. However, these websites used this technique inconsistently and only in authorized cookies browser status(c). In other words, when a user logs into the site, the double submits cookie implements and is only effective in auth. Cookie browser status(c).

Comparison Facebook and Google results

The result for Google and Facebook is quite similar. In the candidate websites, the flow and process of both IdPs are the same as we observed in our

collected data. As follows, we provide the results for google compared to Facebook.

In both Google and Facebook login, 3b(unlinked state: [visitor cookie]) attack strategy has the highest success rate (22.2% for Google and 20.1% for Facebook) in all victim browser statuses. A visitor cookie is the most vulnerable status in both IdPs, which makes the highest success rate compared to other attack strategies, and 3b is the most effective attack strategy. In both IdPs, Test cases 1c and 2c had the lowest detection rates, most probably because targeted websites do not accept new OAuth login when the user is authenticated.

More than 20% of the candidate websites in both IdPs are vulnerable to attack 3. As we concluded in the main draft, OCSRF attacks have a better chance of success in the presence of visitor cookies. With all the similarities in the result for both IdPs, there are some slight differences. Based on our observation, Google mostly uses OpenID Connect + OAuth protocol, which OpenID Connect is used for authentication [51].

The OpenID Connect protocol suggests that websites include a self-signed Nonce to protect against reply attacks. The websites should generate a fresh Nonce, save the Nonce in the browser's cookie and place Nonce in the `return_to` parameter of the OpenID protocol [95].

The site should validate that the Nonce value in the `return_to` URL matches the Nonce stored in the cookie after obtaining the identity assertion from the user's Identity Provider. It ensures that the OpenID protocol session completes on the same browser as it began.

Among 284 websites with Google IdP, only five are using Nonce parameter, and 4 of them are immune to all the attack scenarios. The other one is only exploitable to attack 1b, 3b, and 4b. (visitor cookie: Empty state, Unlinked state, Missing state) and surprisingly not vulnerable to 2b (visitor cookie: lack of state validation) as shown in Table 4.7, classification 10.

The reason might be that the value for both state and nonce parameter in the authorization url is always the same, and any change in the state parameter would raise an error on the server-side. However, our crawler only modifies the state parameter in the authorization url. So we manually tested the attack scenario 2b on this particular site; We modified both state and nonce parameters to the same value, and surprisingly, the site was vulnerable to attack scenario 2b within this small change.

Also, we found one site which uses the constant state and immune to attack scenario 2(lack of state validation) in Table4.7, classification 18, Column 'Const'. The reason might be the request is using the constant state parameter, and any changes also cause an error on the server side. However, the site was vulnerable to other attack scenarios.

In our dataset, 197 websites are deploying both Google and Facebook API, of which 96 websites (48.7%) are immune to OCSRF attack in both IdPs. 38 out of 197 websites (19.3%) are vulnerable and have the same results and pattern in all the attack scenarios.

The remaining websites (63 out of 197 websites) have different patterns for all the attack scenarios. However, in this sample, the number of vulnerable websites with Google IdP is almost twice the ones with Facebook IdP, i.e., there are 25 websites in which only the Google IdP is vulnerable to OCSRF attacks, and 15 websites which are only the Facebook IdP is vulnerable.

Limitations

There are several technologies created especially to find bots and crawlers and to obstruct their operation. We discovered proof of a number of protections used by the websites under examination to avoid automated login and browsing, including CAPTCHA and similar human verification methods.

An attack could fail due to the presence of a properly implemented OCSRF protection or because of the sporadic intervention of a bot detection system.

However, this does not undermine the presented results as they indicate a notably lower bound for vulnerable websites.

False positives could still occur with the tests that run. Our investigation showed that even when the login was incorrectly executed, marker information was occasionally still available. In order to prevent including false positives that were incorrectly deemed as successful in our automated crawler, we manually analyzed every successful attack in order to confirm it.

Another source of errors in testing is the occurrence of temporary service unavailability. Even highly available systems have downtime occasionally, for example, because of network problems, system errors, or planned maintenance. We manually evaluated if these categorization errors were present.

For all the reasons listed above, the list of websites the crawler identified as susceptible is not comprehensive; rather, it merely provides a reasonable range for the number of vulnerable websites among our examined candidate websites, including well-known websites. This demonstrates the need for OCSRF countermeasures and the importance of the implementation errors that our well-considered attack techniques have identified.

4.5 Conclusion

Our work is mainly focused on the analysis of the CSRF attack against `redirect_uri`, a well-known and documented OAuth 2.0 vulnerability.

Our security evaluation found that many real-world SSO service implementations are susceptible to the alleged attack. The difficulty of deploying efficient mitigations and the lack of tools to detect threats accurately are the causes of the predominance of this class of vulnerabilities. We intend to test similar tactics for additional OCSRF attacks in future work.

We created a broad variety of different attack tactics, some of which were innovative, taking into account various implementation flaws and the victim's browser's state at the time of the attack. Our investigation revealed that a number of enabling factors have a significant impact on the attack's viability and contribute significantly to its prevention or success, raising the overall risk.

We looked into a number of the vulnerability's unexplored facets in an effort to cover all the relevant bases and expand our knowledge of how the attack would influence various scenarios. We were able to find various implementation errors and well-hidden vulnerabilities thanks to the enormous number of test cases we carefully analyzed.

Based on the methodology described, we performed a thorough investigation to determine whether OCSRF vulnerabilities existed in 395 sites using two different SSO solutions (i.e., Google and Facebook). For at least one of the planned attack scenarios, more than a third of them were discovered to be weak. This outcome suggests that this security vulnerability still poses a serious challenge for OAuth and OpenID Connect + OAuth authentication systems and that researchers and developers should definitely pay more attention.

Chapter 5

CORS

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. The server is responsible for verifying 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. In the following chapter, we carried out a large-scale analysis of the Tranco Top 50k to measure the prevalence of various implementation flaws due to errors or simplifications in Origin verification.

5.1 Motivation

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 trusted 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 [40, 83, 61].

Unlike previous research, we exploit the vulnerabilities introduced by these CORS flaws in a realistic real-world scenario from the point of view of three attacker models with different capability levels, evaluating the conditions necessary for a successful attack and its consequences. We show how these flaws enable attackers to 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.

5.2 Threat Model

Due to the complexity of the Web, security professionals must consider all the angles and choose suitable attacker models in order to cover possible web attacks. Therefore, as a fundamental assumption of this research, we take three types of attackers from the web security literature: Web attacker, Related-domain attacker, and Network 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 obtains an HTTPS certificate for an arbitrary domain [34]. 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 accordingly to its version. We do not exploit vulnerabilities in the user's browser but only the ones introduced by the CORS flaws on the target websites.

- **Related-domain Attacker**. This attacker is a slightly more powerful web attacker that controls the malicious website hosted on a sibling domain of the target website. A sibling domain is 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 type of attacker is stronger as it is based on simple assumptions of web design, i.e., cookies can be shared between any domain and its siblings, and a related domain attacker can easily compromise the confidentiality and integrity of the target website's cookies [102]. 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. Moreover, an attacker might find an XSS vulnerability on a subdomain, therefore gaining control over the JavaScript code included in the pages on the subdomain.

- **Network Attacker**. These attackers are stronger than the other two attacker models as, in addition to the capabilities of the web and related-domain attackers, they can inspect, spoof, and forge all HTTP traffic on the network. Network attackers can launch man-in-the-middle attacks through, for instance, ARP spoofing and DNS cache poisoning. Network attackers could be the Internet service provider or an attacker in a public WiFi network in a café [34].

5.3 Methodology

We present our measurement methodology in three phases: 1) Collection, 2) Detection, and 3) Exploitation, as shown in figure 5.1.

5.3.1 Phase1: Collection

The first phase aims at finding the web pages to test. We developed a tool based on python and a Selenium web browser to identify the websites using CORS. Our tool checks not only the home pages but also 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 check 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.

As mentioned, SOP and CORS are enforced on domains, and when a website uses CORS, it means the rules are based on the domain. However, a website using CORS shares the resources in some web pages of the website with the same domain, not all. This is the reason we also check for login pages and not only the base domains. If the website has CORS flaws, it will affect the other web pages as well. However, some login pages represent different domains than the website, and their CORS rules might be different.

5.3.2 Phase2: Detection

In this phase, we test all extracted web pages from the previous phase for the CORS flaws listed in Section 2.3. We developed a tool *CORS Flaws Scanner*.

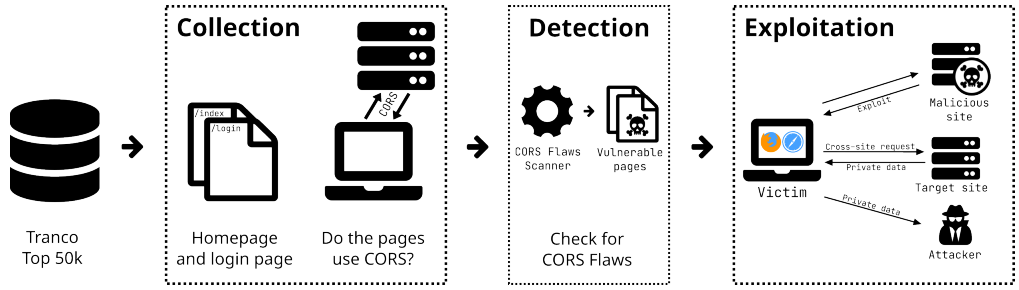

Figure 5.1: High-level visualization of our methodology in three phases: a collection of candidate pages to test, detection of CORS flaws, and flaws exploitation.

Our tool 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.

5.3.3 Phase3: Exploitation

As discussed in Section 2.3, flaws in the way the server checks the value of the Origin header for CORS might lead to security vulnerabilities. In this section, we describe different cross-site attacks that we replicated against several websites with CORS flaws from the perspective of the three attacker models in Section 5.2 and also tested with modern browsers, i.e., Firefox, chrome-based browsers, and Safari.

Web Attacker

To carry out this attack, the web attacker creates a website with a domain name that is mistakenly trusted by the CORS configuration of the target website and generates an HTML page containing some JavaScript code that performs a cross-site request to the target website and has unauthorized

access to the response.

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 has to register the domain using a registrar.

2. The tool creates an exploit HTML containing the JavaScript code presented in Listing 5.1. 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 (that contains sensitive information of the authenticated victim).

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 can generate a certificate trusted by the browser of potential victims using free services such as Let's Encrypt [11]).

4. Finally, the automation uses a puppeteer-controlled browser to open the login page of the target website, where the victim account must log in, and then the victim visits 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 5.1: `URL_PLACEHOLDER` is replaced with the flawed web page on the target website

```
<script>
    var url = 'URL_PLACEHOLDER';
    var req = new XMLHttpRequest();
    req.addEventListener('load', () =>
        console.log(req.responseText));
    req.open('get', url, true);
    req.withCredentials = true;
    req.send();
</script>
```

To exploit the *null_value* flaw, we use a modified JavaScript code that creates an exploit web page where the script 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

Table 5.1: Browsers that default SameSite attribute to Lax when not specified [12]. *Chromium-based include: Chrome, Chromium, Edge, Opera

Browser	Defaults SameSite=Lax
Chromium-based*	Y
Firefox	N
Safari	N

Lax-by-default policy for the SameSite attribute when not otherwise specified (e.g., Firefox, Safari. See Table 5.1).

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 5.3.1), identifies any OAuth button present in the page. Specifically, this tool is based on a Selenium browser and python's *BeautifulSoup* [3] library to crawl through all HTML tags of type *a*, *input* and *button* and checks whether they contain some specific keywords (e.g., "Login 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.

Network Attacker

Instead of registering a domain on the website, a network attacker can intercept the victim's HTTP traffic and inject the malicious JavaScript code directly in a forged response. We replicated the attack on an authenticated victim account as follows:

1. Using a proxy tool, we intercept the victim's HTTP traffic with the hijacked subdomain. Our tool then injects into response the same malicious code used by the web attacker to make an HTTPS request to the main domain. In this way, the Origin header value will be set to the HTTP version of the subdomain, trusted by the target website according to CORS policy.
2. The response is returned to the HTTP subdomain, and it is exfiltrated to the attacker by the malicious JavaScript code.

5.4 Analysis

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 three types of attacker models.

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 on the victims?

(Q3) What can be done by the browser vendors and application developers in order to protect the users' session?

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. Of 6,032 websites using CORS, we found 1,823 (30.2%) 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 three 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 an arbitrary domain for which they possess a valid HTTPS certificate. *2) related-domain attacker*: controls a website hosted on a subdomain of the target website. *3) network attacker*: is the most powerful; can intercept, modify and forge the network traffic and can therefore carry out man-in-the-middle attacks.

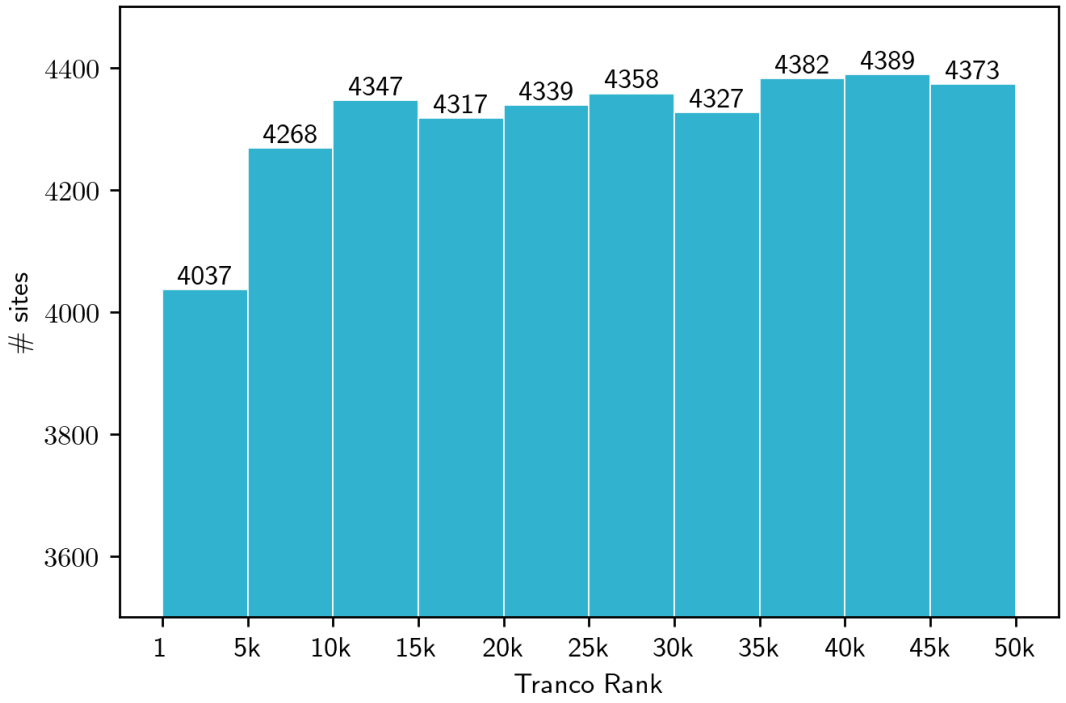

Figure 5.2: Distribution of websites using CORS and misconfigured to at least one variation with respect to their Tranco ranking in 5k bins.

5.4.1 Measurement Overview

We conducted our experiment over the websites included in the Tranco Top 50k [89]. As described in Section 5.3.1, we first crawled each website to identify its homepage and login pages. In total, we identified the homepage of 43,137 websites and the login pages of 17,667 ones. For 6,857 websites, we could not identify the homepage for various reasons, whether the request was timed out or it did not resolve the host with the DNS. In total, we collected and tested 60,804 web pages, only 9,052 of which (on 6,032 different websites) responded with CORS headers.

Using our *Flaws Scanner* tool, we tested all the collected web pages to detect possible CORS flaws. As shown in Table 5.2, we found that 1,823 sites are misconfigured to at least one variation of the ones listed in Section 2.3, for a total of 2,116 web pages.

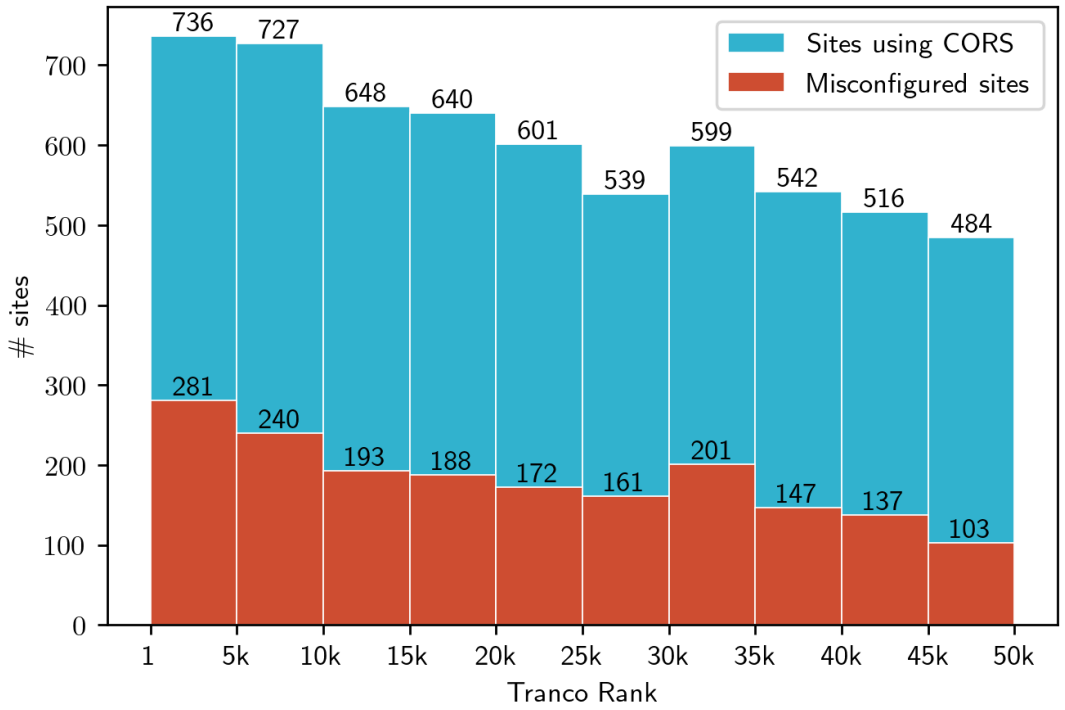

Figure 5.3: Distribution of websites using CORS and misconfigured to at least one variation with respect to their Tranco ranking in 5k bins.

Figure 5.3 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 correlated with the popularity of the website, while the number of websites with CORS flaws is only correlated with the number of websites using CORS. It is important to note that this correlation is not affected by the number of websites for which we could not locate the homepage, which is evenly distributed in the Tranco ranking (except for the first 5k websites, where for almost 1k websites, we could not identify the homepage).

5.4.2 Results

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

Table 5.2: Experiment statistics.

	Tested	Use CORS	Vulnerable
Websites	43075	6032 (14.0%)	1823 (30.2%)
Web pages	60450	9052 (15.0%)	2116 (23.4%)

Table 5.3: Number of sites misconfigured to the tested variations. Percentages are calculated over the number of misconfigured sites (1823).

ID	Variation	Affected sites
1	**HTTPS trusts HTTP**	1625 (89.1%)
2	**Arbitrary subdomain**	762 (41.8%)
3	**Arbitrary origin reflection**	568 (31.2%)
4	**Null value**	553 (30.3%)
5	**Prefix matching**	72 (3.9%)
6	**Suffix matching**	51 (2.8%)
7	**Non escaped dot**	51 (2.8%)

in Section 5.2.

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 website containing the malicious code. As described in Section 5.3.3, for this attack to be possible, the victim must use a browser that does not implement the *Lax-by-default* policy, and three conditions must be met. The variations that meet the first condition are lines 3 to 7 in Table 5.3. Table 5.4 shows the number of websites with CORS flaws that meet the conditions, with a total of 158 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

Table 5.4: Number of sites that meet the conditions required for the attack to be successful for the web attacker listed in Section 5.3.3. The variations referenced in the first conditions are presented in Table 5.3.

Total misconfigured sites	1823
1) Misconfigured to variations 3 to 7	816 (44.8%)
2) Allow credentials	1096 (60.1%)
3) No SameSite attribute	697 (38.2%)
All conditions	**158** (8.7%)

Table 5.5: Number of vulnerable sites and login pages that leak sensitive information of authenticated and unauthenticated victims respectively.

	Authenticated	**Unauthenticated**
Total tested	**30 Sites**	**677 Pages**
Personal Information	16 (53.3%)	—
CSRF Token	10 (33.3%)	155 (22.9%)
CSP Nonce	1 (3.3%)	26 (3.8%)
OAuth State	—	29 (4.3%)

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 5.3.3. We successfully performed the attacks on 24 websites, while for the other six 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 17 websites, categorized in Table 5.5.

Related-domain Attacker

This attacker model can exploit the "Arbitrary subdomain" variation (line 2 in Table 5.3). For this attack to be successful, no further conditions are necessary, and all 762 website users are prone to cross-site attacks.

Network Attacker

This attacker exploits vulnerabilities introduced by the co-presence of the "HTTPS trusts HTTP" and "Arbitrary subdomains" flaws (lines 1 and 2 of Table 5.3). In total, 1,192 websites have both flaws.

Unauthenticated Victim

Motivated by the findings of Mirheidari et al., who in [80] showed how even web pages publicly accessible to unauthorized visitors could 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, presented in Table 5.5. 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 clean browsers. As shown in [32], stealing the state parameter allows attackers to mount successful login CSRF attacks.

5.4.3 Discussion

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 analyzed the conditions necessary for three different attacker models with different 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 analyzed 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% [8], do not implement the *Lax-by-default* policy proposed by Google precisely to mitigate this type of flaw [117]. Firefox initially supported and then discontinued this policy from version 69 in favor of backward compatibility [12]; 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 [4, 2], while for Safari, the intention is not yet clear [5].

Related-domain attackers can exploit one type of flaw, provided he is in control of at least one subdomain of the target website, with severe consequences for the confidentiality of victims' data on the main domain. Finally, with very high prior capabilities, the network attacker can exploit the most common flaw, provided the website also trusts arbitrary subdomains in their HTTP version.

We also answer (Q2) by showing that the exploitation of these vulnerabilities leads to stealing victims' sensitive data. Moreover, we analyzed the consequences of such an attack against unauthenticated victims, stealing the security tokens assigned to them on vulnerable login pages.

As the third question (Q3), we showed that samesite cookies do not directly prevent attackers from accessing cross-origin resources through the CORS flaws, but they do not allow to send the cross-site requests with credentials, little worth to the attacker. To do so, the modern browsers as default or application developers must set the samesite to lax. However, this

setting will not work against other types of attacker models, and with CORS flaws in subdomains and protocols (HTTP, HTTPS), regardless of samesite value, the cookies attached to the cross-site requests.

30% of the websites using CORS had at least one flaw. In particular, almost all websites trust the HTTP version of their domain; nearly half allow arbitrary subdomains, and one in three trusts the value null or simply 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 behaviors or underestimate them. We believe that devolving the burden of verifying the origin programmatically on the server, instead of introducing a policies-based system enforced directly by the browser, has introduced an inherent complexity to the CORS mechanism, which is thus prone to human error or underestimation, 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. During all experiments, we minimized 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, in the coming weeks, we will proceed with the responsible disclosure to all websites that provide a security contact.

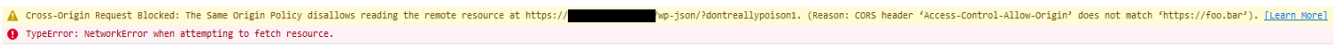

Figure 5.4: CORS error after hitting the cache [6]

5.5 Cache poisoning through CORS

The majority of CDNs do not alter their cache based on random HTTP headers. Therefore, there is a possibility where the cache be poisoned. In a general scenario, if the proper http headers (CORS headers and vary) are not set, then subsequent requests to that URL, with the correct Origin HTTP header will receive the same response from cache. There has been some reports [54, 84] that the Cross-Origin Resource Sharing (CORS) feature in browsers caused a new exploitation by an attacker resulting in a Denial of Service attack on the website.

In this attack, the attacker first sends a request to a website with an origin value. The origin value based on CORS gets reflected in the response and the response will be saved with the "ACAO" header with the first value of origin. Since this vulnerable website does not set the 'vary' header, this response in cache will be served to the requests to this host due to hitting the cache first to get the response. So when other users try to initiate a request to this host with different origin value, they receive the error shown in figure 5.4.

However, not all the websites are programmed to cache their response. If the response contains some http headers regarding to Cache, there is a high possibility that they cache the response, such as 'Cache-Control', 'X-Cache' (based on the CDN the headers would be different). For instance, in our dataset, we found that 1,294 out of 1,823 websites (71%) cached their responses. For a successful DoS/CORS attack one condition must be met, the 'vary=origin' not be set in the response header and in our measurements, 491 out of 1,294 websites (38%) did not set this header, which make a huge number of websites to be vulnerable to this type of attack by simply not

setting a 'vary' header.

5.6 Conclusion

A web security tool called the Same-Origin Policy (SOP) enables limiting access to resources on a website from origins other than the site itself. The three values of protocol, host, and port together identify the origin of a website. However, the SOP could be excessively restrictive and undermine the functioning of a website if it depends on exchanging data with third-party websites of different sources.

The Cross-Origin Resource Sharing (CORS) mechanism was established for websites that seek to continue cross-site information exchange with specific third-party websites without giving up the usage of the SOP as a protection mechanism.

In order to respond to cross-site requests, CORS relies on two HTTP headers: "Access-Control-Allow-Origin" (ACAO), which lets the server specify whether to trust the origin of the request and "Access-Control-Allow-Credentials" (ACAC), which lets the server specify whether authentication cookies and any authorization headers may be attached to requests by the browser.

While the server is in charge of determining whether or not to trust requests after validating their origin and their value, the client browser is responsible for enforcing the rules set out by CORS. Because of this, the logic that confirms the value of the origin is essential for the website's security.

Due to the fact that CORS origin verification is programmed by the application developers, there is a considerable risk of introducing defects that cause users to trust websites that may be under the control of hostile actors, jeopardizing the security of the website.

The most straightforward instance of a risky CORS configuration is when

the origin value of the request is simply copied into the ACAO header of the response, thereby trusting every potential origin. Errors in the construction of regular expressions, the use of prefixes or suffixes in the checks, or allowing the value null can all result in other problematic settings.

We first conducted a large-scale analysis on the homepages and login pages of the websites in the Tranco Top 50k to measure the prevalence of several variations of CORS implementation flaws.

We found 1,823 out of 6,032 (30.2%) websites using CORS having 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 three 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 an arbitrary domain for which they possess a valid HTTPS certificate.

2) related-domain attacker: controls a website hosted on a subdomain of the target website.

3) network attacker: is the most powerful; can intercept, modify and forge the network traffic and can therefore carry out man-in-the-middle attacks.

We discovered that the consequences of the exploitation of CORS flaws could 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 subsequent attacks (such as CSRF and login CSRF).

We also identified possible additional conditions necessary for CORS flaws to be exploited by attackers against victims using modern browsers with default security settings. We developed a methodology to exploit candidate websites with CORS flaws and replicate the attacks in a realistic real-world scenario. We analyzed the consequences of exploiting CORS flaws and dis-

cussed potential solutions to mitigate them.

Chapter 6

Conclusion

I have taken OAuth protocol (vastly studied on its security) and CORS mechanism (newly introduced and rapidly adapted) into account for this thesis. We investigated the feasibility and effectiveness of novel approaches to measure and reduce the security risks introduced by current patches for web applications and their users.

Even though these two concepts have been introduced to protect their users' privacy while increasing the Web's functionality, as shown in this thesis, in real-world scenarios, the goals have not been appropriately achieved due to many factors and elements involved.

In chapter 4, a wide range of attack strategies and scenarios have been implemented by a repeatable methodology on large-scale OAuth deployments in the wild. As a result, one-third of high-profile websites are still vulnerable to CSRF (the most studied attack).

Analysis showed that not only the primary mitigation (presence of `state`) did not work in all the situations, but also different cases, deactivated other CSRF mitigations, even the most effective one (user interaction), showing how inconsistent these mitigations are in an interaction with each other.

In chapter 5, we provided a large-scale study on CORS implementations in the wild by considering all the elements present in every situation (e.g.,

threat models, modern browsers) and propose a semi-automated methodology to perform the attacks regarding these security issues in order to find the involving factors to successful exploitation.

We showed that each attacker model with a different level of capabilities could endanger the users' privacy and security. We looked into each attack scenario's current and active mitigations in every situation. We proposed some solutions to decrease the danger of these mistakes in high-profile web applications.

Of course, for both studies and the development of the methodologies, we faced many limitations. Nevertheless, we overcame most of them through novel approaches and achieved more test cases leading to accurate results.

Security professionals can use these repeatable, automated methodologies to perform a large-scale measurement for OAuth protocol and CORS mechanism to discover the well-hidden vulnerabilities in real-world implementations.

Bibliography

[1] RFC 6454. https://www.rfc-editor.org/info/rfc6454, author = Adam Barth, title = The Web Origin Concept, pagetotal = 20, year = 2011, month = dec,.

[2] 1617609 - (samesitelax) [meta] Enable sameSite=lax by default. RE-OPENED (n.goeggi) in Core - Networking: Cookies., 2022-07-15, https://bugzilla.mozilla.org/show_bug.cgi?id=1617609.

[3] Beautiful Soup Documentation — Beautiful Soup 4.9.0 documentation. https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

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

[5] Cookies default to SameSite=Lax - Chrome Platform Status. https://chromestatus.com/feature/5088147346030592.

[6] Cors'ing a denial of service via cache poisoning. https://nathandavison.com/blog/corsing-a-denial-of-service-via-cache-poisoning.

[7] Fastly documentation. configuring caching, 2021. https://docs.fastly.com/en/guides/configuring-caching.

[8] Global Desktop Browser Market Share for 2022. https://kinsta.com/browser-market-share/, language = en-US, urldate = 2022-07-01, journal = Kinsta®,.

[9] James kettle. practical web cache poisoning. portswigger web security blog, 2018. https://portswigger.net/blog/practical-web-cache-poisoning.

[10] James kettle. web cache entanglement: Novel pathways to poisoning. portswigger research, 2020. https://portswigger.net/research/web-cache-entanglement.

[11] Let's Encrypt. https://letsencrypt.org/, abstract = Let's Encrypt is a free, automated, and open certificate authority brought to you by the nonprofit Internet Security Research Group (ISRG)., 2022-06-28,.

[12] SameSite cookies - HTTP | MDN. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite, abstract = The SameSite attribute of the Set-Cookie HTTP response header allows you to declare if your cookie should be restricted to a first-party or same-site context.,2022-06-28,.

[13] Samesite cookies csrf attacks. https://symfonycasts.com/screencast/api-platform-security/samesite-csrf.

[14] Webspi and web application models. http://prosecco.gforge.inria.fr/webspi/. Accessed: 2019-08-30.

[15] Akamai developer. edgeworkers, 2021. https://developer.akamai.com/akamai-edgeworkers-overview.

[16] Builtwith technology lookup., 2021. https://trends.builtwith.com/CDN/Content-Delivery-Network.

[17] Cloudflare docs. cloudflare workers documentation., 2021. https://developers.cloudflare.com/workers/.

[18] Fastly. compute@edge., 2021. https://www.fastly.com/products/edge-compute/use-cases.

[19] Referrer-policy, 2021. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy, 2021.

[20] Omer Akgul, Taha Eghtesad, Amit Elazari, Omprakash Gnawali, Jens Grossklags, Daniel Votipka, and Aron Laszka. The hackers' viewpoint: Exploring challenges and benefits of bug-bounty programs.

[21] Devdatta Akhawe, Adam Barth, Peifung E Lam, John Mitchell, and Dawn Song. Towards a formal foundation of web security. In 2010 23rd IEEE Computer Security Foundations Symposium, pages 290–304. IEEE, 2010.

[22] Tamjid Al Rahat, Yu Feng, and Yuan Tian. Oauthlint: an empirical study on oauth bugs in android applications. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 293–304. IEEE, 2019.

[23] Wade Alcorn, Christian Frichot, and Michele Orru. The Browser Hacker's Handbook. John Wiley & Sons, 2014.

[24] Marios Argyriou, Nicola Dragoni, and Angelo Spognardi. Security flows in oauth 2.0 framework: a case study. In International Conference on Computer Safety, Reliability, and Security, pages 396–406. Springer, 2017.

[25] Elham Arshad, Michele Benolli, and Bruno Crispo. Practical attacks on login csrf in oauth. Computers & Security, page 102859, 2022.

[26] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. Authscan: Automatic extraction of web authentication protocols from implementations. In NDSS, 2013.

[27] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Keys to the cloud: formal analysis and concrete attacks on encrypted web storage. In International Conference on Principles of Security and Trust, pages 126–146. Springer, 2013.

[28] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis 1. Journal of Computer Security, 22(4):601–657, 2014.

[29] Adam Barth. HTTP State Management Mechanism. RFC 6265, April 2011. https://www.rfc-editor.org/info/rfc6265.

[30] Adam Barth, Collin Jackson, and John C Mitchell. Robust defenses for cross-site request forgery. In Proceedings of the 15th ACM conference on Computer and communications security, pages 75–88, 2008.

[31] Adam Barth, Collin Jackson, and John C Mitchell. Securing frame communication in browsers. Communications of the ACM, 52(6):83–91, 2009.

[32] Michele Benolli, Seyed Ali Mirheidari, Elham Arshad, and Bruno Crispo. The Full Gamut of an Attack: An Empirical Analysis of OAuth CSRF in the Wild, pages 21–41. Springer International Publishing, Cham, 2021.

[33] Bruno Blanchet et al. An efficient cryptographic protocol verifier based on prolog rules. In csfw, volume 1, pages 82–96, 2001.

[34] Andrew Bortz, Adam Barth, and Alexei Czeskis. Origin cookies: Session integrity for web applications. ACM Transactions on Internet Technology (TOIT), 2, 05 2012.

[35] Stefano Calzavara, Riccardo Focardi, Matteo Maffei, Clara Schneidewind, Marco Squarcina, and Mauro Tempesta. {WPSE}: Fortifying web protocols via browser-side security monitoring. In 27th {USENIX} Security Symposium ({USENIX} Security 18), pages 1493–1510, 2018.

[36] Brian Campbell, Michael Jones, and Chuck Mortimore. Security assertion markup language (saml) 2.0 profile for oauth 2.0 client authentication and authorization grants. 2015.

[37] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In Proceedings 42nd IEEE Symposium on Foundations of Computer Science, pages 136–145. IEEE, 2001.

[38] Suresh Chari, Charanjit S Jutla, and Arnab Roy. Universally composable security analysis of oauth v2. 0. IACR Cryptology ePrint Archive, 2011:526, 2011.

[39] Eric Y Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. Oauth demystified for mobile application developers. In Proceedings of the 2014 ACM SIGSAC conference on computer and communications security, pages 892–903. ACM, 2014.

[40] Jianjun Chen, Jian Jiang, Haixin Duan, Tao Wan, Shuo Chen, Vern Paxson, and Min Yang. We still Don't have secure Cross-Domain requests: an empirical study of CORS. In 27th USENIX Security Symposium (USENIX Security 18), pages 1079–1093, Baltimore, MD, August 2018. USENIX Association.

[41] James Clark, Steve DeRose, et al. Xml path language (xpath), 1999.

[42] Alexei Czeskis, Alexander Moshchuk, Tadayoshi Kohno, and Helen J Wang. Lightweight server support for browser-based csrf protection. In Proceedings of the 22nd international conference on World Wide Web, pages 273–284, 2013.

[43] Neil Daswani, Christoph Kern, and Anita Kesavan. Cross-domain security in web applications. Foundations of Security: What Every Programmer Needs to Know, pages 155–196, 2007.

[44] Shehroze Farooqi, Fareed Zaffar, Nektarios Leontiadis, and Zubair Shafiq. Measuring and mitigating oauth access token abuse by collusion networks. In Proceedings of the 2017 Internet Measurement Conference, pages 355–368, 2017.

[45] Daniel Fett, Ralf Küsters, and Guido Schmitz. An expressive model for the web infrastructure: Definition and application to the browser id sso system. In 2014 IEEE Symposium on Security and Privacy, pages 673–688. IEEE, 2014.

[46] Daniel Fett, Ralf Küsters, and Guido Schmitz. Spresso: A secure, privacy-respecting single sign-on system for the web. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pages 1358–1369. ACM, 2015.

[47] Daniel Fett, Ralf Küsters, and Guido Schmitz. A comprehensive formal security analysis of oauth 2.0. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 1204–1215. ACM, 2016.

[48] R. Fielding. Hypertext transfer protocol – http/1.1. RFC 2616, 1999. https://datatracker.ietf.org/doc/html/rfc2616.

[49] Mohammad Ghasemisharif, Amrutha Ramesh, Stephen Checkoway, Chris Kanich, and Jason Polakis. O single {Sign-Off}, where art thou? an empirical analysis of single {Sign-On} account hijacking and session management on the web. In 27th USENIX Security Symposium (USENIX Security 18), pages 1475–1492, 2018.

[50] Kevin Gibbons, John O Raw, and Kevin Curran. Security evaluation of the oauth 2.0 framework. Information Management and Computer Security, 22(3), 2014.

[51] Google Developers. Using oauth 2.0 to access google apis, 2020. https://developers.google.com/identity/protocols/oauth2.

[52] Chong Guan, Kun Sun, Lingguang Lei, Pingjian Wang, Yuewu Wang, and Wei Chen. Dangerneighbor attack: Information leakage via postmessage mechanism in html5. Computers & Security, 80:291–305, 2019.

[53] Run Guo, Jianjun Chen, Baojun Liu, Jia Zhang, Chao Zhang, Haixin Duan, Tao Wan, Jian Jiang, Shuang Hao, and Yaoqi Jia. Abusing cdns for fun and profit: Security issues in cdns' origin validation. In 2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS), pages 1–10. IEEE, 2018.

[54] Run Guo, Weizhong Li, Baojun Liu, Shuang Hao, Jia Zhang, Haixin Duan, Kaiwen Sheng, Jianjun Chen, and Ying Liu. Cdn judo: Breaking the cdn dos protection with itself. In NDSS, 2020.

[55] Y GURT. Critical issue opened private chats of facebook messenger users up to attackers., 2013. https://www.bugsec.com/news/facebook-originull/,2013,2022-06-28.

[56] HackerOne. Hackerone bug bounty platform, 2020. https://www.hackerone.com/.

[57] E Hammer-Lahav. The oauth 2.0 authorization protocol. draft-ietf-oauth-v2-16, 2011.

[58] D. Hardt. The oauth 2.0 authorization framework. RFC 6749, 2012. http://www.rfc-editor.org/rfc/rfc6749.txt.

[59] Dick Hardt. The oauth 2.0 authorization framework. 2012.

[60] Egor Homakov. The most common oauth2 vulnerability. Technical report, 2012. http://homakov.blogspot.com/2012/07/saferweb-most-common-oauth2.html.

[61] Evan J. Misconfigured cors, 2016. https://ejj.io/misconfigured-cors,2022-06-28.

[62] Ed. J. Reschke. Hypertext transfer protocol (http/1.1): Semantics and content. RFC 7231, 2014. https://datatracker.ietf.org/doc/html/rfc7231.

[63] Martin Johns and Justus Winter. Requestrodeo: Client side protection against session riding. In Proceedings of the OWASP Europe 2006 Conference, 2006.

[64] Florian Kerschbaum. Simple cross-site attack prevention. In 2007 Third International Conference on Security and Privacy in Communications Networks and the Workshops-SecureComm 2007, pages 464–472. IEEE, 2007.

[65] James Kettle. Exploiting cors misconfigurations for bitcoins and bounties, 2016. https://portswigger.net/research/exploiting-cors-misconfigurations-for-bitcoins-and-bounties.

[66] Soheil Khodayari and Giancarlo 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), May 2022.

[67] Apurva Kumar. Using automated model analysis for reasoning about security of web protocols. In Proceedings of the 28th Annual Computer Security Applications Conference, pages 289–298, 2012.

[68] Sebastian Lekies, Walter Tighzert, and Martin Johns. Towards stateless, client-side driven cross-site request forgery protection for web applications. SICHERHEIT 2012–Sicherheit, Schutz und Zuverlässigkeit, 2012.

[69] Frank Li, Zakir Durumeric, Jakub Czyz, Mohammad Karami, Michael Bailey, Damon McCoy, Stefan Savage, and Vern Paxson. You've got vulnerability: Exploring effective vulnerability notifications. In 25th {USENIX} Security Symposium ({USENIX} Security 16), pages 1033–1050, 2016.

[70] Wanpeng Li and Chris J Mitchell. Security issues in oauth 2.0 sso implementations. In International Conference on Information Security, pages 529–541. Springer, 2014.

[71] Wanpeng Li, Chris J Mitchell, and Thomas Chen. Mitigating csrf attacks on oauth 2.0 and openid connect. arXiv preprint arXiv:1801.07983, 2018.

[72] Wanpeng Li, Chris J Mitchell, and Thomas Chen. Oauthguard: Protecting user security and privacy with oauth 2.0 and openid connect. In Proceedings of the 5th ACM Workshop on Security Standardisation Research Workshop, pages 35–44, 2019.

[73] Soheil Khodayari Giancarlo Pellegrino. Likaj, Xhelal. Where we stand (or fall): An analysis of csrf defenses in web frameworks. In 24th International Symposium on Research in Attacks, Intrusions and Defenses., 2021.

[74] Labunets Fett Lodderstedt, Bradley. draft-ietf-oauth-security-topics-15. Technical report, 2020. https://tools.ietf.org/html/draft-ietf-oauth-security-topics-15.

[75] Ziqing Mao, Ninghui Li, and Ian Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In International Conference on Financial Cryptography and Data Security, pages 238–255. Springer, 2009.

[76] Mark Higgins. Symantec internet security threat report, 2003. https://docs.broadcom.com/doc/istr-03-jan-en.

[77] MDN. Preflight requests in cors, 2022. https://developer.mozilla.org/en-US/docs/Glossary/Preflight_request.

[78] Gordon Meiser, Pierre Laperdrix, and Ben Stock. Careful Who You Trust: Studying the Pitfalls of Cross-Origin Communication. In ASIACCS 2021 - 16th ACM Asia Conference on Computer and Communications Sec 16th ACM Asia Conference on Computer and Communications Security, Hong Kong / Virtual, China, June 2021.

[79] Seyed Ali Mirheidari, Sajjad Arshad, Kaan Onarlioglu, Bruno Crispo, Engin Kirda, and William Robertson. Cached and confused: Web cache deception in the wild. In 29th {USENIX} Security Symposium ({USENIX} Security 20), pages 665–682, 2020.

[80] Seyed Ali Mirheidari, Matteo Golinelli, Kaan Onarlioglu, Engin Kirda, and Bruno Crispo. Web cache deception escalates! In 31st USENIX

Security Symposium (USENIX Security 22), Boston, MA, August 2022. USENIX Association.

[81] Mitre. Url redirection to untrusted site, 2020. https://cwe.mitre.org/data/definitions/601.html.

[82] Vladislav Mladenov, Christian Mainka, and Jörg Schwenk. On the security of modern single sign-on protocols: Second-order vulnerabilities in openid connect. arXiv preprint arXiv:1508.04324, 2015.

[83] Jens Müller. On Web-Security and -Insecurity: CORS misconfigurations on a large scale, July 2017. https://web-in-security.blogspot.com/2017/07/cors-misconfigurations-on-large-scale.html.

[84] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. Your cache has fallen: Cache-poisoned denial-of-service attack. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pages 1915–1936, 2019.

[85] OAuth.net. User authentication with oauth 2.0. Technical report, 2020.

[86] OWASP. Cookies: Http state management mechanism draft-ietf-httpbis-rfc6265bis-02, 2021. https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-rfc6265bis-02#section-5.3.7.

[87] OWASP. Cross-site request forgery prevention, 2021. https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html.

[88] Suhas Pai, Yash Sharma, Sunil Kumar, Radhika M Pai, and Sanjay Singh. Formal verification of oauth 2.0 using alloy framework. In 2011 International Conference on Communication Systems and Network Technologies, pages 655–659. IEEE, 2011.

[89] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In Proceedings 2019 Network and Distributed System Security Symposium. Internet Society, 2019.

[90] David Recordon and Brad Fitzpatrick. Openid authentication 2.0-final. Network working group internet-draft, 2007. https://openid.net/specs/openid-authentication-2_0.html.

[91] G. REVAY. Here it is, the file upload csrf 2013. http://gerionsecurity.com/2013/04/here-it-is-the-file-upload-csrf/.

[92] Leonard Richardson. Beautiful soup. Crummy: The Site, 2013. https://www.crummy.com/self/resume.html.

[93] Philippe De Ryck, Lieven Desmet, Thomas Heyman, Frank Piessens, and Wouter Joosen. Csfire: Transparent client-side mitigation of malicious cross-domain requests. In International Symposium on Engineering Secure Software and Systems, pages 18–34. Springer, 2010.

[94] Philippe De Ryck, Lieven Desmet, Wouter Joosen, and Frank Piessens. Automatic and precise client-side protection against csrf attacks. In European Symposium on Research in Computer Security, pages 100–116. Springer, 2011.

[95] Nat Sakimura and John Bradley. Openid connect core 1.0 incorporating errata set 1. 2014.

[96] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. Same-Origin policy: Evaluation in modern browsers. In 26th USENIX Security Symposium (USENIX Security 17), pages 713–727, Vancouver, BC, August 2017. USENIX Association.

[97] Robin Sharma. Preventing cross-site attacks using same-site cookies, 2017. https://dropbox.tech/security/preventing-cross-site-attacks-using-same-site-cookies.

[98] Ethan Shernan, Henry Carter, Dave Tian, Patrick Traynor, and Kevin Butler. More guidelines than rules: Csrf vulnerabilities from non-compliant oauth 2.0 implementations. In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pages 239–260. Springer, 2015.

[99] Jaimandeep Singh and Naveen Kumar Chaudhary. Oauth 2.0: Architectural design augmentation for mitigation of common security vulnerabilities. In Journal of Information Security and Applications 65, 2022.

[100] Kapil Singh, Alexander Moshchuk, Helen J Wang, and Wenke Lee. On the incoherencies in web browser access control policies. In 2010 IEEE Symposium on Security and Privacy, pages 463–478. IEEE, 2010.

[101] Sooel Son and Vitaly Shmatikov. The postman always rings twice: Attacking and defending postmessage in html5 websites. In NDSS, 2013.

[102] Marco Squarcina, Mauro Tempesta, Lorenzo Veronese, Stefano Calzavara, and Matteo Maffei. Can i take your subdomain? exploring Same-Site attacks in the modern web. In 30th USENIX Security Symposium (USENIX Security 21), pages 2917–2934. USENIX Association, August 2021.

[103] Marius Steffens and Ben Stock. Pmforce: Systematically analyzing postmessage handlers at scale. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pages 493–505, 2020.

[104] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. How the web tangled itself: Uncovering the history of {Client-Side} web ({In) Security}. In 26th USENIX Security Symposium (USENIX Security 17), pages 971–987, 2017.

[105] Ben Stock, Giancarlo Pellegrino, Christian Rossow, Martin Johns, and Michael Backes. Hey, you have a problem: On the feasibility of large-scale web vulnerability notification. In 25th {USENIX} Security Symposium ({USENIX} Security 16), pages 1015–1032, 2016.

[106] Avinash Sudhodanan, Alessandro Armando, Roberto Carbone, Luca Compagna, et al. Attack patterns for black-box security testing of multi-party web applications. In NDSS, 2016.

[107] Avinash Sudhodanan, Roberto Carbone, Luca Compagna, Nicolas Dolgin, Alessandro Armando, and Umberto Morelli. Large-scale analysis & detection of authentication cross-site request forgeries. In 2017 IEEE European symposium on security and privacy (EuroS&P), pages 350–365. IEEE, 2017.

[108] Karin Sumongkayothin, Pakpoom Rachtrachoo, Arnuphap Yupuech, and Kasidit Siriporn. Overscan: Oauth 2.0 scanner for missing parameters. In International Conference on Network and System Security, pages 221–233. Springer, 2019.

[109] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: An empirical analysis of oauth sso systems. In Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12, pages 378–390, New York, NY, USA, 2012. ACM.

[110] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: an empirical analysis of oauth sso systems.

In Proceedings of the 2012 ACM conference on Computer and communications security, pages 378–390, 2012.

[111] San-Tsai Sun, Yazan Boshmaf, Kirstie Hawkey, and Konstantin Beznosov. A billion keys, but few locks: the crisis of web single sign-on. In Proceedings of the 2010 New Security Paradigms Workshop, pages 61–72. ACM, 2010.

[112] Ed. T. Lodderstedt. Oauth 2.0 threat model and security considerations. RFC 6819, 2018. https://www.rfc-editor.org/rfc/rfc6819.txt.

[113] David Y Wang, Stefan Savage, and Geoffrey M Voelker. Cloak and dagger: dynamics of web search cloaking. In Proceedings of the 18th ACM conference on Computer and communications security, pages 477–490, 2011.

[114] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In 2012 IEEE Symposium on Security and Privacy, pages 365–379. IEEE, 2012.

[115] Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, and Yuri Gurevich. Explicating sdks: Uncovering assumptions underlying secure authentication and authorization. In 22nd {USENIX} Security Symposium ({USENIX} Security 13), pages 399–314, 2013.

[116] Mike West. Incrementally better cookies. (2019). https://tools.ietf.org/html/draft-west-cookie-incrementalism-00.

[117] Mike West. Incrementally Better Cookies. Internet-Draft draft-west-cookie-incrementalism-01, Internet Engineering Task Force, March 2020. Work in Progress.

[118] WHATWG. Fetch Standard. https://fetch.spec.whatwg.org/,2022-06-29.

[119] John Wilander. CORS-safelisted request headers should be restricted according to RFC 7231 · Issue #382 · whatwg/fetch. https://github.com/whatwg/fetch/issues/382.

[120] Damon Williams. Introduction to paypal. Pro PayPal E-Commerce, pages 1–12, 2007.

[121] Luyi Xing, Yangyi Chen, XiaoFeng Wang, and Shuo Chen. Integuard: Toward automatic protection of third-party web service integrations. In NDSS, 2013.

[122] Guangliang Yang, Jeff Huang, Guofei Gu, and Abner Mendoza. Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications. In 2018 IEEE Symposium on Security and Privacy (SP), pages 742–755. IEEE, 2018.

[123] Ronghai Yang, Wing Cheong Lau, Jiongyi Chen, and Kehuan Zhang. Vetting single sign-on {SDK} implementations via symbolic reasoning. In 27th {USENIX} Security Symposium ({USENIX} Security 18), pages 1459–1474, 2018.

[124] Ronghai Yang, Wing Cheong Lau, and Tianyu Liu. Signing into one billion mobile app accounts effortlessly with oauth2. 0. blackhat Europe, 2016.

[125] Ronghai Yang, Guanchen Li, Wing Cheong Lau, Kehuan Zhang, and Pili Hu. Model-based security testing: An empirical study on oauth 2.0 implementations. In Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, pages 651–662, 2016.

[126] Xiaofeng Zheng, Jian Jiang, Jinjin Liang, Haixin Duan, Shuo Chen, Tao Wan, and Nicholas Weaver. Cookies lack integrity:{Real-World} implications. In 24th USENIX Security Symposium (USENIX Security 15), pages 707–721, 2015.

[127] Yuchen Zhou and David Evans. Ssoscan: automated testing of web applications for single sign-on vulnerabilities. In 23rd {USENIX} Security Symposium ({USENIX} Security 14), pages 495–510, 2014.

[128] Ningxian Zhu. Security of cors on localstorage. In 2021 International Conference on Internet, Education and Information Technology (IEIT), pages 141–146. IEEE, 2021.