Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science

Master's Thesis

Investigating the Impact of Persistent State on Client-Side CSRF in Web Applications

Secure Web Applications Group, CISPA

submitted by

Shubham Agarwal

on 30. April 2020

Reviewers

Dr.-Ing. Ben Stock

Prof. Dr. Christian Rossow

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Saarbrücken, 30. April 2020

(Shubham Agarwal)

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Saarbrücken, 30. April 2020

(Shubham Agarwal)

Statement

I hereby confirm the congruence of the contents of the printed data and the electronic version of the thesis.

Erklärung

Ich erkläre hiermit, dass die vorliegende Arbeit mit der elektronischen Version übereinstimmt.

Saarbrücken, 30. April 2020

(Shubham Agarwal)

Abstract

The web applications on the Internet have steadily evolved over the years and aims to provide services to their users that were infeasible a decade ago, hence, keeping in pace with the advancements in web technologies. Integration of third-party libraries, cross-domain interaction & increased client-side code and data storage enable these applications to extend their functionalities. At the same time, this often leads to heightened exploit surface, and exposure to critical threats such as XSS and command injection. CSRF is one such threat to web applications which the research community has broadly addressed over time and proposed countermeasures to defend against it. However, we observe that these vulnerabilities are reported in the wild every once in a while.

This thesis addresses a new window of CSRF vulnerabilities that arise from the usage of persistent storage APIs available to store stateful data at client-side and its resultant impact on the server-side state of the application. It presents the first principle characterization and framework to conduct a broad study on this class of vulnerability in CSRF, termed as „Client-Side CSRF“. We show that these exploit surfaces may exist on the web and be further abused by an adversary even when traditional CSRF mitigation is in place.

In the first part of this thesis, we present a detailed description of the proposed vulnerability and implement an automated framework to determine the existence of these vulnerabilities among real-world web applications. We also discuss the threat models which an attacker may leverage to exploit this vulnerability. While, in the second part, to determine the prevalence of Client-Side CSRFon the current web, we investigate over top 1 million domains to find these issues and further simulate exploits for them based on the existence of flows. We report that over 13.9% of these domains store data at the client-side storage, that it later uses to perform potentially sensitive operations at the server. Further, we show that 2.98% of all these domains are exploitable by an adversary, as verified by our proposed framework, thus, affirming the existence of the threat surface.

Acknowledgments

I wish to acknowledge all those people who have been kind and supportive to me through the course of my thesis work. Firstly, I would like to thank my thesis advisor Dr.Ing Ben Stock for providing me with an astounding opportunity as well as an interesting problem statement to work upon under his supervision. His constant support and guidance to students and passion for research have been truly inspirational for me. I would also like to thank Prof. Christian Rossow for agreeing to review my thesis as the second reviewer.

My next thanks go to Marius Steffens who has been tremendously patient and insightful every single time I needed his advice. He has been the go-to person for all the doubts and supported me to extend his previous work in this thesis. I also wish to extend my gratitude to the whole Secure Web Applications Group for their friendly support and fruitful discussions.

I would also like to thank all my friends who have been supportive, patient and a persistent source of motivation. Last, but never the least, I sincerely thank my family for their constant and unconditional support and encouragement.

Contents

Chapter 1

Introduction

An age-old web attack, *Cross-Site Request Forgery (CSRF)*, has been brought down to its feet due to numerous effective countermeasures proposed by researchers and implemented by the application developers. However, we believe that a new exploit vector in the form of *Storage APIs* may be abused by the adversaries to forge requests even when traditional CSRF countermeasures are in place. This thesis intends to draw the attention of the research community towards this new window of vulnerability at the client-side of the web applications. In this chapter, we briefly discuss the advancement of web applications in recent times, the threats that surround them and the research statement of this work.

1.1 Motivation

Over the years, the Internet has become an indispensable agent in various aspects of day to day lives, affecting more than half of the entire population of the world. Web applications hosted on the Internet has evolved consistently with time and technological progress. These applications provide a varied range of services such as online education, e-banking, e-commerce, entertainment, auction, voting, social media & communication, to name a few. It also facilitates business enterprises to effectively reach out to their target audience other than providing with a collaborative platform.

Furthermore, the applications, nowadays, are inclined to be more user-centred and serve dynamically-generated content customized for different users. The increased client-side capability of web applications enables powerful features such as PWAs, Push Notifications and Offline Resource Availability & Synchronization, at user's disposal. These features intend to not only help reduce server-side processing and data storage but also generate and customize content on-the-go. The complexity of these applications is further aided by the third-party service providers, to facilitate auxiliary services at the client such as payment, tracking and advertisement [1, 2]. For instance, in many applications, the embedded third-party *chat-bots* have predefined responses and operations usually generated and stored at the client storage [3].

Along with the utility of web applications, their security is of paramount importance and has become a crucial factor in their development in recent times. Adding powerful functionalities at client-side also often leads to increased logic and data storage at the client, as reported in [4]. Moreover, one cannot neglect the fact that these technologies may also attribute to wider exploit surface for malicious entities on the web. Cyber Security incidents reported in the past indicate that the web is susceptible to even decade-old attacks like *SQL Injection*, *Broken Authentication* and *Cross-Site Scripting* and they still dominate the top 10 threats listed by OWASP, although there are numerous countermeasures proposed by the researchers in [5–13]. For instance, *ClearView AI*, an AI-based facial-recognition enterprise, recently reported data theft of 3 Billion private photos scrapped from social media profiles due to broken authentication at the client resulting in large-scale privacy leak [14].

At the heart of client-side vulnerabilities, lies the fact that most of the times, exploits occur due to untrusted data flow from the client to the server. The web adversary can bypass the input validation and sanitization methods as they are either flawed or implemented incorrectly by the developer [15–17]. Furthermore, there often exists an inherent trust in the integrity of data stored on the client by the application server [18]. The data flow from the client-side storage APIs, therefore, adds to the exploit surface if the integrity of the data goes unverified at the server such that it may be under the influence of an adversary. These vulnerabilities may lead to large-scale repercussions and thus, needs to be addressed thoroughly by the research community as well as application developers to avoid unwanted consequences.

1.2 Problem Statement

Cross-Site Request Forgery, also known as CSRF, XSRF or session-riding, is one of the oldest yet powerful attack, and has dominated as one among the OWASP top 10 Web Applications Vulnerabilities for over a decade. The adversary, in this case, intends to issue requests to the application server, that seems to be benign and originally issued by the user from within the application. These state-changing request may have serious implications ranging from fraudulent bank transaction, posing fake statuses on social media, changing passwords, account deletion or complete account takeover, depending upon the context of the application and the access-privilege of the user exploited in the attack.

In this research study, we assume that an adversary can potentially influence the client-side data of the target web application. We then demonstrate that if a request from the client to the server carries malicious data flow from the client-side storage, it may cause state-changing actions at the server-side. Thus, this backdoor provides an adversary with the opportunity to forge requests issued by the client along with untrusted data flow to the server, even when traditional CSRF mitigation techniques work as required.

To this end, we intend to investigate the type of data stored in different storage entities available at the client and further track their flow to the application server on each page visit under both normal and adversarial conditions. Additionally, we also intend to observe the influence of such an adversary over cross-origin communication using *Message APIs* that occur between frames embedded on the parent application. Based on the identified flows, we describe this new vulnerability window in CSRF, termed as „Client-Side CSRF“. We discuss a few specific cases of vulnerabilities in applications and report our findings.

1.3 Thesis Outline

The outline of the thesis is structured as follows. In chapter 2, we discuss prior research contributions by the web security community. In Chapter 3, we describe the technical definition of different client-side technologies relevant to this work as well as the traditional CSRF vulnerability. Further, in Chapter 4, we introduce in detail the concept of Client-Side CSRF, its relevance in today's web landscape and differences when compared to traditional CSRF attack. In Chapter 5, we present our automated research framework to detect and exploit Client-Side CSRFvulnerabilities amongst web application in the wild. Based on our methodology defined in Chapter 5, we evaluate the collected data from the framework implemented over the top 1 million domains in Chapter 6. We then present our overall results from the analysis and evaluation of the existence of vulnerabilities and discuss a specific case in Chapter 7. Finally, in Chapter 8, we review generic validation mechanisms deployed by a few applications and discuss the diverse nature of sources and sinks concerning Client-Side CSRFvulnerabilities. We also mention the limitations of this thesis and potential future work before concluding in Chapter 9.

Chapter 2

Related Works

In this chapter, we review different investigations conducted by the security researchers over time, which coincide with our study. We begin by outlining the works which describe the threats that exist due to insecure usage of the client-side storage. We then discuss recent works done to detect and mitigate CSRF attacks. Lastly, we also mention other recently discovered client-side vulnerabilities on the web relevant to this study.

2.1 Issues with Persistent Storage at the Client

Ever since the introduction of Web Storage APIs in 2010, there have been numerous works addressing the lack of integrity in client-side storage. Hanna *et al.*first reported the vulnerability in this client-side storage such as WebSQL and localStorage in [19]. They conducted a broad study over 11 applications to manually analyze the use of Web APIs and further reported that the developers put an inherent trust on these storage containers and lacks data sanitization before re-use leading to persistent XSS backdoors. Contrastingly in this work, we perform a large-scale empirical study to analyze the use of stateful data stored at the client to perform state-changing actions at the server.

Lekies and Johns [18] first conducted a large scale study in 2012 to determine the usage of persistent storage APIs by web applications. They reported code caching patterns into localStorage, providing an exploit surface for injection and XSS vulnerability for attackers. To verify the integrity of data stored in these storage containers, they also propose a minimal wrapper functionality. However, other recent works suggest that while the solution seems feasible to avoid integrity issues, not many web applications in the wild have deployed it.

Zheng et al. [20] demonstrated that the cookies are vulnerable to injection and hence, lacks integrity. Sivakorn et al. [21] further highlight this issue, proving that an attacker could inject or extract data stored in the cookies. Thus, storing sensitive data in cookies may lead to security implications when used by the client at a later stage. However, in this work, we examine the impact of the stateful data stored in cookies which could be under the adversarial influence and later processed by the server.

Steffens et al. [22] conducted an end-to-end empirical study over 5,000 domains to detect vulnerable flows that originate from client-side storage and could be exploited by an attacker to perform Persistent Client-Side XSS. They showed that over a significant number of domains use cookies and localStorage to store data which later flows to security-critical JavaScript sinks such as *eval* without any integrity-check or data sanitization. However, in this work, we build an end-to-end framework to identify the vulnerable data source stored at the client-side storage when later used to perform the state-sensitive operation by the application server.

2.2 Cross-Origin Attacks in the Wild

Though first being reported back in 2002 [23], CSRF has drawn notable attention from the security community continuously due to the severity of the threat it poses on the web. In early works to detect and mitigate CSRF attacks, the authors in [24–26] demonstrate the prevalence of such issues and its subsequent impact of exploitation, based on the vulnerabilities reported until then. They also proposed various defensive strategies to tackle against CSRF such as HTTP-referrer validation, origin validation and custom-headers, to identify the origin of the request at the server. De Ryck et al. [27] recommended a heuristic-based domain whitelisting approach to allow third-party requests while Pelizzi and Sekar [28] suggested an additional server-side proxy to intercept and validate the origin of the request based on secret tokens. However, we observe that though these mitigation strategies prove to be effective in many cases, they are often bypassed by unintended backdoors or implemented incorrectly by the application developers.

Sudhodanan et al. [12] showed that a large number of top-ranked domains are vulnerable to authenticated-CSRF attacks which highlight the absence or improper usage of CSRF mitigation techniques. Recently, Pellegrino et al. [29] proposed a first automated security framework to detect the CSRF vulnerabilities on the web. They leveraged code-property graphs described in [11] to model dynamic information traces generated by the PHP application and identify security-sensitive state transitions. This model is then used to identify vulnerable CSRF candidates in real-world applications. Calzavara et al. [30] in their recent work used ML-based approach for black-box detection of CSRF vulnerabilities. They performed supervised learning to train their model on known sensitive and insensitive requests. The model learns to distinguish request based on different heuristics and further identify CSRF vulnerable candidates in real-world applications.

Contrasting to works in [29, 30], we specifically focus on the end-to-end dynamic analysis of stateful data flows to the server, which originates from the client-side storage as second-order vulnerabilities Dahse and Holz [31].

2.3 Other Associated Client-Side Vulnerabilities

Our study intersects with other client-side security issues addressed in the past by the research community. In [8], Lekies et al. used taint-tracking browsing engine to identify data flows exposed to DOM-Based XSS and showed that over 9% of the top 5K domains are vulnerable to such attacks. In their following works, as in [9, 32], Stock, Lekies and others investigated the root-cause for the DOM-based XSS discovered in their previous work and also additionally analyzed the efficiency of XSS-filtering mechanism at the client for these vulnerabilities. They further reported the dangers associated with dynamic generation and inclusion of JavaScript code into the application from cross-origin or third-party libraries in [13], known as XSSI.

Nikiforakis et al. [33] investigated top 10K domains to study the security impact of the inclusion of third-party resources in the parent application. They highlighted four different vulnerabilities based on their analysis and proposed suitable countermeasures for them. Acker et al. further proposed least privilege integration policies to restrain the execution environment and avoid cross-origin vulnerabilities arising from the third-party resources included, as in [1]. Stock et al. [4], investigated the evolution of web over time using Internet Archives and reported that increased client-side complexity in terms of logic, third-party script inclusions and cross-domain data access over the time also provides exploit surface to a multitude of attacks such as XSS and content-sniffing, aligning with the results of previous works. Recent work by Ikram et al. [34] shows that while most of the websites render resources from at least three different sources, this chain of trust may extend up to 30 distinct domains.

In [35], Son & Shmatikov investigated the usage of the PostMessage API to allow cross-origin communications in application by relaxing the same-origin policy. They conduct their study on 10K distinct websites and further state that the application developers often use this API without necessary origin-check at the receiving-end leading to the backdoor for exploits such XSS & content-injection. They proposed origin-check and CSP as countermeasures. Guan *et al.*further hinted at the abuse of the PostMessage API for eavesdropping on messages communicated by different origins in [36] and proposed similar countermeasures as in [35].

Other works by different researchers such as in [2, 37–39] have demonstrated numerous security and privacy issues concerning the integration of third-party content and cross-domain communication. We consider the vulnerabilities discussed above to assess the impact of a web attacker who can, for instance, execute a malicious script or inject arbitrary data in the context of the application for our proposed vulnerability in this work.

Chapter 3

Technical Background

In this chapter, we briefly discuss different data storage APIs available and used at the client to understand and categorize the data stored based on their persistence. We then discuss the mechanism by which the client may communicate the stored data with its application server or third-party service providers. Further, we review our understanding of traditional CSRF vulnerability along with numerous mitigation strategies proposed by web security researchers over the years.

3.1 Client-Side Storage

The web applications hosted over the Internet run over *Hypertext Transfer Protocol (HTTP)* (or *HTTPS*) protocol. HTTP is stateless by design such that it does not retain any session information from the communication that takes place between the client and the server [40]. In this case, the client, as well as the server, needs to store the session data for further communication. Moreover, there exist certain client-specific functionalities that execute at each page visit and doesn't change across sessions. For example, language preference or cart details may remain the same across sessions for a given user in an application. Instead of sending this information to the client on every visit, the server may store them on the client to avoid redundant processing. Thus, these features require some storage mechanism at the client where the server can store data specific to the session of the application or the user, analogous to the databases at server-side. We describe a few of them relevant to this work as follows.

3.1.1 Cookies

Cookies were introduced by Lou Montulli from Netscape Communications, back in 1994, to address session management issue at the client for stateful applications, since the HTTP is stateless by design [41–43]. They are nothing but key-value pairs stored and maintained by the browser at client-side. The client can set the cookie for the session using JavaScript while the server can do the same using the „*Set-Cookie*“ header sent along with the response. *Session cookies* exist until an active session and get deleted by the browser later. However, cookies may persist even after the session is concluded, by explicitly setting the expiration date & time for them, called *persistent cookies*. Cookies with *HttpOnly* attribute are not accessible to the client using JS but only by the server when sent along with the request. Authentication tokens, for instance, are often stored in *HttpOnly* cookies.

Cookies are restricted to the domain of an application, meaning that cookies set for a particular domain could only be available to that domain, including its subdomains. For example, cookies set for `.foo.com` could be accessible only to `foo.com` itself as well as `sub.foo.com`, while it would not be accessible to other domains such as `bar.com`. While setting cookies, the current domain may specify its domain and can also set the cookie for its parent domain, e.g. `sub1.foo.com` can set cookies for `foo.com`, accessible by `foo.com`, `sub1.foo.com`, `sub2.foo.com` and all other subdomains of `www.foo.com`. By default, the browser restricts cookies to the scope of domain setting it, depending upon their fail-safe defaults [44]. These cookies are accessible across subdomains irrespective of the fact that whether or not they run under HTTPS in case the parent domain accepts the request only on a secured connection.

When the client requests the server, the browser tags along all the cookies available for that domain to indicate the *session state* of the client to the server, often termed as authentication cookies. However, cookies serve a lot more purpose on the modern web other than just session management. For instance, tracking cookies gather the browsing history of the user and reports back to the server. These cookies are either first-party cookies set by the top-level host accessed by the user, or third-party cookies set by any other domain enclosed within the hostname [45].

3.1.2 Persistent Storage

We discussed the ways client stores the session state, and further sends along each of the requests to the server for authentication and access, as discussed above in Section 3.1.1. There are several other application-wide data and configurations which don't change frequently and span across sessions, contrary to the session-specific data. Application personalisation data such as language preferences, themes, shopping cart details and browsing history are few of the examples. Cookies don't appear to be the right fit to store such data, even though it could „persist" after the session, due to several limitations. The browser vendors only allow to set the minimum required number of cookies per domain, as specified in RFC 2965 (as well as in RFC 6265). Moreover, in most of the browsers, only 4096 bytes of data could be stored within each cookie.

3.1.2.1 Web Storage

To eliminate the above-mentioned limitations of cookies and persist more data at the client, the HTML WHATWG proposed another storage medium, called *Web-Storage* [46, 47]. The Web-Storage is accessible by *Storage APIs* exposed by the browser. Similar to cookies, web-storage also allows storing data in the form of key-value pairs. The data could be stored and accessed by specifying the key using exposed *setter* and *getter* APIs, respectively. Further, at least 5 MB of data could be stored per origin in the web-storage depending upon the browser used, and the browser does not implicitly send the storage data to the server, unlike cookies.

While the cookies are bound just to the domain, the web-storage is bound to the origin of the document. The origin constitutes a tuple of *{scheme, domain, port}*. For instance, a cookie set for the origin `https://foo.com:80` could be accessed by `http://foo.com:88`, however, data stored in the web-storage for the origin `https://foo.com:80` would not be accessible on `http://foo.com` or `https://foo.com:88`.

The Web Storage provides two different storage options depending upon the persistence of the data required by the application: *sessionStorage* & *localStorage*. The data stored in sessionStorage last as long as the page session exists and gets flushed away automatically when the browser closes. However, opening multiple instances of the application in different window result in individual storage entry for each window. LocalStorage retains the data for as long as it is not explicitly removed from the browser using JavaScript or by manually clearing the store. Both of these storage containers have respective APIs to store and fetch data. In this work, we only focus on localStorage and skip sessionStorage since data stored in localStorage persists across sessions, but that's not the case with sessionStorage.

3.1.2.2 IndexedDB

To address the requirements of client-side features such as offline editing of documents, push messaging and PWAs, there occurred a need to store larger and more complex chunk of data other than just string at the client [48]. Not until recently, the W3C proposed a transaction-based database API, called the IndexedDB. IndexedDB is designed as an object-oriented database, accessible using JavaScript. It allows storing 50 MB of data per origin.

The storage structure of the IndexedDB resembles server-side databases such that there may exist multiple databases for an origin, and each database contains object store(s), analogous to the tables in SQL. The IndexedDB also exposes an API to create indexes on the stored data other than setting and getting data from the database. The data stored inside an object store is in the fixed-column structure as key and value, which could be a simple string or complex JSON document. The database handles each of the operations asynchronously by treating them as individual transactions. The IndexedDB is also bound to the origin of the page, as in web-storage.

3.2 Communication - Within & Across Origin

The client communicates with its application server on same-origin by *XMLHttpRequest (XHR) API* using JavaScript. It allows two-way communication between the two endpoints without reloading the document. Recently developed frameworks such as *AJAX* and *Fetch API* are built on top of XHR and further abstracts the complexity and allow asynchronous processing.

However, several security directives restrict cross-origin communication on the web, as discussed in [37, 49–51]. For instance, *Same-Origin Policy (SOP)* [52] controls the interaction between documents hosted on different origins, i.e., a parent document of origin *https://accounts.foo.com* cannot read the response from the request issued to the server hosted at *https://cdn.foo.com* as the origin of the two endpoints don't match. One way to achieve this is to configure CORS headers at all communication endpoints.

To enable cross-origin communication by relaxing the SOP at client, HTML5 exposes messaging API, called *PostMessages* [35, 53]. It allows communication between frames or windows with different origins at client-side by obtaining the reference of the target window and further sending the message. The receiving window listens to the message event and handles the received data accordingly. The receiver may further reply to the source of received message event using `event.source.postMessage` as illustrated in [54].

3.3 Cross-Site Request Forgery

In this attack, the adversary aims to destroy the integrity of the existing authenticated session a user has already established with the target application vulnerable to CSRF, termed as authenticated-CSRF or aCSRF. There exists another variant of CSRF, called login-CSRF, where the attacker forces the victim to log-in to the target application using attacker-credentials. However, we only consider aCSRF for this work. In this case, the attacker forces another legitimate user to perform certain actions for them without directly communicating to the server, a case of confused-deputy. We describe the steps of this attack in Figure. 3.1 as follows:

1. Firstly, the attacker lures the user to the attacker domain by using either social-engineering techniques such as emails or providing overt services to the user.
2. The user visits the attacker-controlled domain.
3. Upon visiting this domain, the attacker delivers the maliciously crafted request pointing to the target application server as the payload.
4. When the browser parses the payload sent by the attacker, it inadvertently issues the request to the target application in the context of the established session for the user.

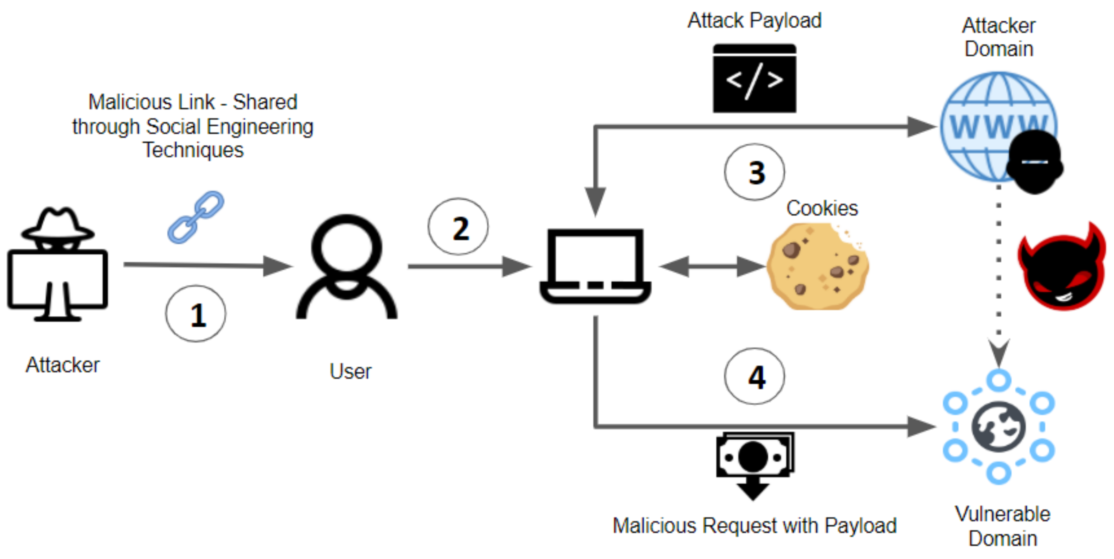
Attack Payload
Attacker Domain
Malicious Link - Shared through Social Engineering Techniques
3
Cookies
2
1
Attacker
User
4
Malicious Request with Payload
Vulnerable Domain

Figure 3.1: Traditional CSRF Attack

Thus, the adversary forges a malicious request and then forces the user to send it to the server within the context of an authenticated session. The payload could be a simple image embedded on the attacker domain with malicious source as shown in Listing 3.1 or a form containing malicious data to be posted to the target server as in Listing 3.2. The browser tags along with all the cookies as request headers and send it to the server. Upon receiving the request, the server further processes it and perform the desired action, assuming that the request is genuinely issued.

This vulnerability is considered as a server-side flaw as the server is unable to discriminate between genuine and forged requests. There are numerous CSRF mitigation techniques proposed by the researchers [12, 25, 28], few of which we discuss as follows:

a) ***Anti-CSRF Tokens*:** These are secret nonces generated by the server on each page load, which it forwards to the client after storing its copy with itself. The client embeds this secret along with all sensitive requests as hidden input for origin verification at server-side. Since the client store the nonce at client-storage, the attacker cannot extract or manipulate the token without bypassing SOP.

b) ***Double-Submit Cookies*:** In this technique, the server remains stateless as it does not store the secret token generated for the client. Instead, it sends

```
<img src="http://foobank.com/transfer?amount=1000&accountTo=Eve"/>
```

Listing 3.1: Payload example as iframe embedded on attacker domain

```
<form method="POST" name="transfer" action="http://foobank.com/transfer>
  <input type="hidden" name="accountTo" value="Eve">
  <input type="hidden" name="accountFrom" value="Alice">
  <input type="hidden" name="amount" value="1000">
</form>
document.forms.transfer.submit();
```

Listing 3.2: Payload example as Form POST

the original secret as cookie data to the client. Upon receiving the request, the server identifies the actual origin by comparing the secret embedded in the request data against the copy stored in the cookie data. This technique again relies on the fact that the attacker cannot manipulate cookies.

c) ***Same-Site Cookies*****:** The *„Same-Site“* attribute is introduced for cookies to mitigate CSRF that arises from the default behaviour of sending all the cookies set for the target domain. It can have either of the three values: *Strict*, *Lax* or *None*. *Strict* value instructs the browser to send no cookies associated with the target domain along with the requests that originate from a third-party domain. *Lax* value allows the browser to send cookies only along with those request which makes top-level navigation to the target domain and the request type is *GET*. When the value is *None*, the browser tags along all the cookies set for the target domain while sending the request, irrespective of the domain of the sender. The default value is either set to *Lax* or *None*, depending upon the browser.

Chapter 4

Client-Side CSRF

In this chapter, we discuss in detail the concept of Client-Side Cross-Site Request Forgery. We describe the ways by which persistent data stored at the client could be abused by an adversary to successfully forge malicious requests even when CSRF countermeasures are in place. We further discuss the ways by which an attacker may be able to influence the data at client storage. Lastly, we outline the differences that exist between traditional CSRF & the proposed Client-Side CSRF.

Before discussing the impact of storing stateful data at the client-side storage, we first define the notion of *state* for this work. While the state in a web application generally refers to the session state of the user at the server, we do not intend to study the session state here. Depending on the nature of the data and the context in which the application uses it to perform sensitive operations at the server, we classify the state as either as *application state* or as *user state*, stored at the client-side.

The application state refers to the data that a website store at the client usually when the user visits it for the first time and uses it on every subsequent visit to tailor the content as per users' preference. This state persists until the user explicitly wipes the browser cache and is user-independent. Thus, it doesn't require the user to login. Skin mode, geographical location and language may serve as examples for application state. The user state refers to the data or settings stored by the application-specific to the user currently logged in to the application. Depending upon the website, the user state may supersede the application state. This state persists on the client-side unless acted upon by the client-side application or the user explicitly clears up the browser cache, similar to the application state.

The examples for user state may include user-specific cookie-consent and policies, subscription status and currency preference.

4.1 Persistent CSRF Vulnerability at Client

To perform any state-changing action at the target application, the attacker needs to forge malicious request, similar to the one legitimately issued by the victim. However, if mitigation strategies such as *anti-csrf tokens* defends the application against CSRF, they would not be able to retrieve the secret token without access to the client-storage of the target domain. We assume that this is not possible for an active attacker due to the *Same-Origin Policy* enforced by the browser. Due to the given nature of the threat, we also observe that even when there are no secret tokens in place, the attacker needs to craft the payload and deliver it to the user every time they wish to exploit the vulnerability.

However, we argue that if there exists a data flow which originates from the persistent data store at the client and then sent to the server without any integrity-check to execute security-critical operations, the application may be susceptible to this attack. More precisely, instead of creating the entire malicious request as a payload, the adversary only requires to modify the data used to form genuine requests by the client, and thus it can also be considered as „persistent“ Client-Side CSRF. The exploit surface is broader in case the payload is stored on persistent cookies and flows to the server as these cookies are not bound to the origin but the domain, also exposing the subdomains to this vulnerability. Similarly, if any of the subdomains is under the attacker's control, it could set cookies for the parent domain along with the payload.

We demonstrate this vulnerability with the help of a hypothetical example as follows. A social network website, *mysocial.net*, accepts and processes all the XHRs on a dedicated server, *api.mysocial.net*. The request header contains application-wide anti-csrf token used to authenticate the request at the server, which refreshes after every use. The application stores this dedicated endpoint at the client, as shown in Listing 4.1. Now, for every request the user makes, the browser retrieves the XHR endpoint from the storage to send the request. We show that an adversary who can influence the client-data stored can also manipulate this XHR endpoint as per his choice. As demonstrated in Listing 4.2, when the user intends to view

```
<script>
  function onPageVisit() {
    let xhr_endpoint = window.location.hash.substr(1);
    //xhr_endpoint = "api.mysocial.net";
    localStorage.setItem("xhr_api_domain", xhr_endpoint);
  }
</script>
```

Listing 4.1: Example usage of stateful data stored in localStorage

```
<script>
  function getProfilePic(userId) {
    //Assuming that the attacker has already altered the storage data.
    let url = "https://" + localStorage.getItem("xhr_api_domain") +
    ↪  "/profilePic?user=userId";
    //Expected Link: https://api.mysocial.net/profilePic?user=userId

    //Actual Link: https://api.mysocial.net/postStatus?post='Hello!'
    //            &ru=https://api.mysocial.net/profilePic?user=userId
    sendRequest(url);
  }
</script>
```

Listing 4.2: Example usage of localStorage data sent to perform sensitive operation at the server

the profile picture, the client fetches the endpoint URL and makes the request. Since, the attacker, in this case, modifies the endpoint such that whenever the user makes a request, it would first post a status on the user's wall and then perform the desired operation as specified in *returning url* parameter (*ru*). This way, the attacker can persist as well as force the user to execute the payload without violating the anti-CSRF countermeasures. It is a specific instance where the dedicated endpoint serves as the application state, and the attacker modifies it to post content of his choice on the victim's wall. However, this exploit vector could allow to perform other critical operations as well, such as profile deletion, add or remove connections.

As above, there are a lot more scenarios where the application would want to store data at the client, only to re-use it later. Cookie consent policies, affiliate and tracking identifiers could be few of the examples. For instance, IndexedDB is heavily used by applications to store large documents for editing, when offline [55]. These changes are synchronized with the server when the user is online the next time. The server can verify the integrity of the received data only if it has generated it. However, there is no means to verify the data generated by the client

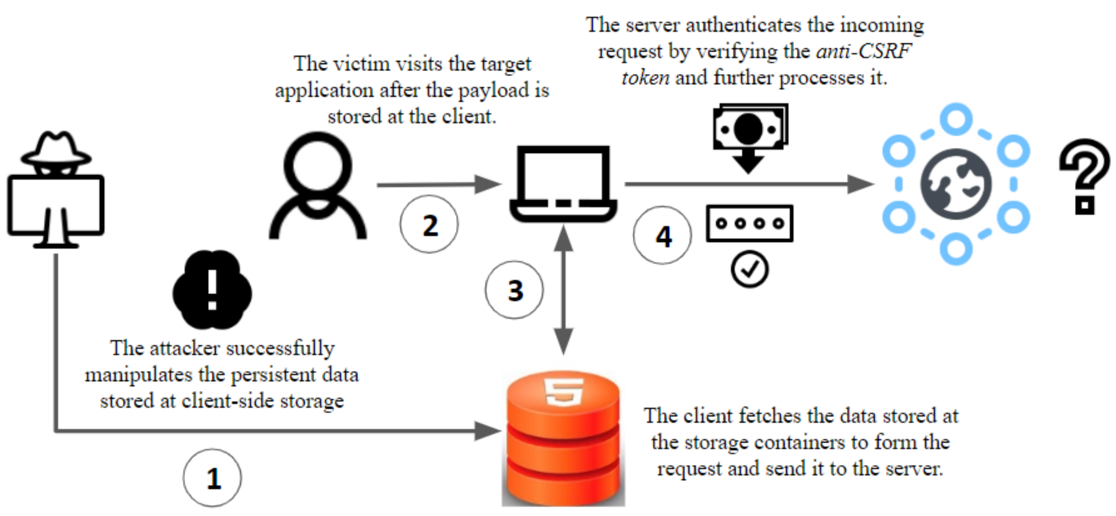
The server authenticates the incoming request by verifying the *anti-CSRF token* and further processes it.
The victim visits the target application after the payload is stored at the client.
2
4
3
The attacker successfully manipulates the persistent data stored at client-side storage
The client fetches the data stored at the storage containers to form the request and send it to the server.
1

Figure 4.1: Client-side CSRF - Vulnerable data flow that originates from the persistent web storage and sent to the server to perform sensitive operation.

using JavaScript or by the third-party service providers, embedded within the parent application, later stored at the client.

In general, we claim that any stateful-data stored at the client, which flows to the server to execute the sensitive operation is subject to investigation for this study. As shown in Figure 4.1, we describe the source to sink data flow considered for this study as follows:

1. The attacker identifies the data source at the client later used by the application to perform sensitive action on the server and injects the payload for its persistence. For instance, they can replace the original value for the target key with any preferred value.

2. After the attacker store the payload at the victim's client, the victim then visits the target application endpoint and perform the operation which involves attacker-controlled data flow. It does not necessarily mean that the victim needs to establish a new session for the attacker to exploit.

3. When the application issues a request that involves the target data flow, the browser fetches the attacker-controlled value this time to form the request and further send this to the server.

4. Upon receiving the request, assuming that the CSRF countermeasures are in place, the server authenticates the request by verifying the anti-csrf token.

Since the client issues the request from within the application, it contains the correct hidden token, and the authentication succeeds.

We identify the localStorage, IndexedDB & persistent cookies as the persistent data source for our study. While localStorage allows storing data which span across multiple sessions, introduction and usage of IndexedDB further promote the applications to store even larger and complex data at the client for which there exist no implicit means to erase them. Persistent cookies do no less damage, as even though it stores a smaller chunk of data, it may still be sufficient enough to carry stateful information such as user preferences and session secrets. Further, we consider the XMLHttpRequest & PostMessage APIs as data sinks which facilitates the same-origin and cross-origin communication, respectively, as also described in Section 3.2.

4.2 Attackers' Capabilities

To influence the requests sent to the server, the adversary should have control over the data that gets stored at the client. We model the adversary's capabilities in line with the two well-established threat models also discussed by Steffens et al. [22]. We summarize the attacker model and prerequisites to manipulate the data at the client as follows:

4.2.1 Network Attacker

A Man-in-the-Middle or network attacker may inject, drop or manipulate packets within an unencrypted communication channel that exists between the client and the server. However, the attacker would only be limited to domains accepting connections on HTTP, and not on HTTPS domains [56] since they would not be able to obtain a valid TLS certificate for communication. So, in this case, we limit our adversary's capacity only over HTTP connections [21].

The attacker can inject an HTML page in the response body or set arbitrary headers of its choice. Cookies can be manipulated by tampering or injecting the cookie headers into the packets that flow as a response from the server or by using JavaScript inserted with the HTML page. Moreover, since the cookies are bound to the domain, a non-HTTPS sub-domain can set cookies for parent domain running on HTTPS. This way if the victim visits the target domain or sub-domain over unencrypted connection even for once, the attacker may inject the payload and make it persist at the client for exploitation later when connected on HTTPS. The inserted HTML page can further point to another target domain for which it would able to set cookies. However, we notice that the domains with HTTP Strict-Transport Security header configured prevents the users from establishing any unencrypted connection channel.

As described above, the attacker can similarly inject the HTML page to set payload into the localStorage and IndexedDB for the target application. While the attacker may be able to influence the data flowing into localStorage or IndexedDB by injecting HTML into the response over an unencrypted communication, they are unable to control the storage of target domain from any of their subdomains since they are bound to the origin, unlike cookies. Thus, this disallows any non-HTTPS communication channel to modify the storage data stored for its HTTPS counterpart.

We consider all those domains as potentially vulnerable, which accepts the connection over HTTP and further contains the source to sink flow discussed in Section 4.1. We also take into account those domains which allow unencrypted communication on any of their subdomains even when they communicates on HTTPS. Further, we consider those domains which employ HSTS configuration, yet are not included in HSTS preload list [57] or skips setting the *includeSubDomains* modifier [58–60], leaving the subdomains vulnerable.

4.2.2 Web Attacker

In this threat model, instead of modifying the data that gets transmitted over the network, the attacker forces the user to visit an attacker domain under her control by several social-engineering techniques [61–63]. For instance, a link to an attacker website providing seemingly benign content, or spam emails with deceptive content may serve the purpose. This attacker-controlled domain may contain document elements, such as iframe or anchor elements, heading to the target application. Once the user visits this attacker site, the browser parses the malicious element and issues the request to the target, along with the attacker-defined payload. This could lead to exploitation in the two following cases:

a.) The URL parameters may flow to either of the data sinks directly or indirectly via client-storage. In case when these parameters are the malicious payloads specified by the attacker on its domain, unfiltered usage of this data may lead to exploitation. However, it is necessary for the attacker to precisely identify the URL parameters (and storage keys in case it flows through the web-storage) which the client uses to carry out sensitive operations later.

b.) The attacker can leverage the existence of other web vulnerabilities at the client, such as Reflected or Persistent Client-Side Cross-Site Scripting, to inject malicious payload at the client. For instance, the attacker can inject malicious JS to store payload into any of the persistent storage containers. If this payload further flows to insecure XSS sinks, such as *eval*, it may execute JavaScript and store this payload which persists across sessions. In this way, they do not need the user to visit the attacker-domain the next time they wish to carry out the same attack.

In addition to the above-stated threat models, we note that the adversary in this vulnerability does not violate any anti-CSRF countermeasures which are deployed by the application. We consider the web applications vulnerable to this attacker model when it either consists of client-side XSS vulnerabilities or contains URL parameters to XMLHttpRequest (or PostMessage) flow and thus, the application is also vulnerable to Client-Side CSRF.

4.3 Difference From Traditional CSRF

The attacker model described above, as well as other recent research works, have provided significant evidence to believe that an attacker can compromise the integrity of the client-side storage and further use it as an attack vector to perform malicious actions on the target application in case no proper data validation and sanitization occurs. We also saw how an adversary could exploit the long known version of CSRF in a target application server and outlined the countermeasures that have proved to be successful in curbing this attack on the web. Nevertheless, we believe that the vulnerability window proposed in this work would persist and the attacker could exploit it, bypassing the existing CSRF countermeasures at the client. Traditional CSRF mitigation strategies like *anti-csrf tokens* or *double-submit cookies* intend to prevent the server from processing those malicious requests which are concocted by an adversary and do not originate at the client. It is because the secret token used to form the request at the client would be missing since the adversary would not be able to retrieve it from outside of the application.

However, in the case of Client-Side CSRF, the adversary doesn't need to craft malicious request outside the application. Instead, they can influence the persistent data stored at the client, which the client later uses to create authentic request upon the victim's action. More precisely, the attacker crafts the payload only once and inject it into the client-storage. This payload flows to the server along with every succeeding request to the target application endpoint. In this way, the attacker doesn't have to guess or extract the secret token to forge a request. Furthermore, since the manipulated data persist across multiple sessions, the adversary can exploit without any additional effort every time this data flows to perform the target operation. Although, the attacker still doesn't get to access the response of the issued request to the server, much like traditional CSRF.

Chapter 5

Research Methodology

So far, we discussed about the definition of Client-Side CSRF, along with its association with traditional CSRF vulnerabilities. We also outlined the threat model which an attacker may resort to exploiting the target domain. In this chapter, we present our automated framework to detect these vulnerabilities in web applications hosted over the Internet. We outline the goal of this framework before discussing its technical specifications. The proposed framework intends to find answers for the following research questions in line with the vulnerability as follows:

1. How many application uses the data stored in persistent storage containers at the client to perform state-changing operations at the server?
2. In how many applications, can the above data flow be abused by the adversary given that they can control the data stored at the client-side storage?
3. Further, how many of the above application can be successfully exploited to perform state-changing operations within the scope of the attackers' capabilities discussed in Section 4.2?

Based on the vulnerable source to sink data flow discussed in 4.1, we recognise our data source as the client-side storage entities, i.e. cookies, localStorage and IndexedDB. We then identify our sinks that carry requests away from the client, i.e. XMLHttpRequests & PostMessages. It is important to note that the data sources and sinks are not itself dangerous as they can perform benign and state-insensitive

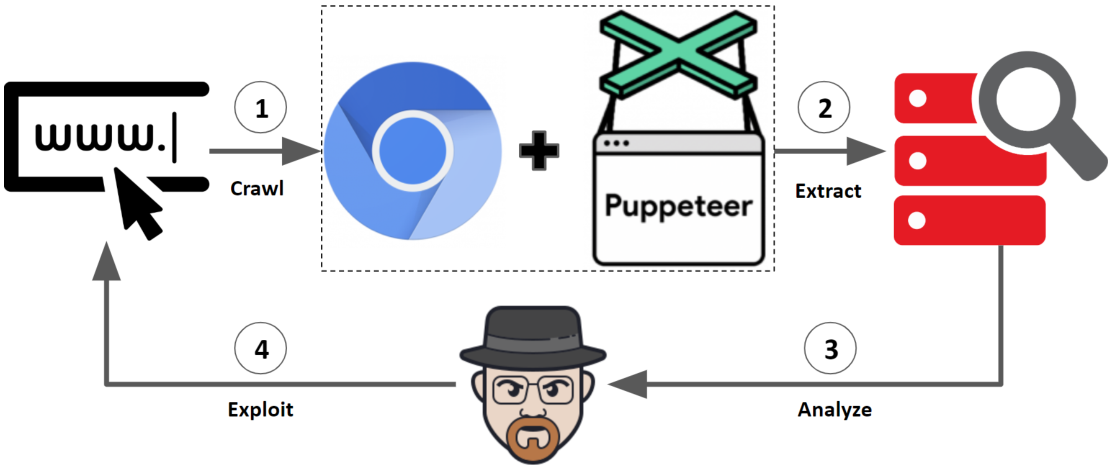

Figure 5.1: An Overview of the Methodology

functionalities. It is the insecure and unfiltered use of the sensitive data which provides an exploit surface to the attacker when the integrity of data stored in these persistent containers are compromised. Based on previous large-scale vulnerability detection works in [11, 12, 64, 65], we develop our understanding and describe our overall framework, as shown in Figure 5.1.

5.1 Data Flow Collection

We build a chrome extension to collect the data that gets stored into one of the sources and then further flows to either of the sinks. To gather all the data that the application stores into localStorage, we overwrite the setter and getter APIs. We use *Object.defineProperty* [66] in JavaScript to overwrite the native definition used to set and get data into localStorage and add our logging functionality. The new proxy setter and getter collects the storage key and value as is and store it in our database before storing it into the respective storage as desired by the application.

Similarly, we overwrite APIs used to store and retrieve data to and from cookies, respectively. Overwriting the native APIs instead of retrieving cookies from JavaScript using *document.cookie* API [67] provides an additional advantage. It also reports the *HttpOnly* cookies which would otherwise not be accessible by the latter API. We also intercept all the responses received by the application in the extension background to extract *Set-Cookie* headers [68] and log its values. Then, we hook all the setters and getters for IndexedDB in a similar manner. In this case, there are multiple getter API which an application could use to retrieve data [69]. For instance, *getall* retrieves all the values from the given object-store while *get* takes the key as an input parameter and only returns the value associated with the key. For getters, we only log the key and not the value retrieved because of the asynchronous nature of the IndexedDB.

To listen to all the postmessage events that get dispatched to the embedded frame, and vice versa, we overwrite the required *contentWindow* properties of the HTMLIFrameElements [70], such that proxy handlers extract the message along with its source and target before performing the desired operation. Similarly, we overwrite the source definition of *MessageEvent API* [71] to listen to the messages sent as a reply to the original postmessage by the target window. We further log all the outgoing requests from the application by intercepting them outside the application environment using the NodeJS framework, puppeteer, also used to instantiate the crawling session. Lastly, we extract all the links embedded on the visited page to track the flow of the URL parameters to either of the data sinks. Thus, we collect all the data that gets stored as well as the sink operations performed upon them to confirm the data flow.

The contentScript injects the script to the parent page along with its iframes to overwrite the target APIs at *document_start* [72]. It means that our script is included to the page before any other script executes and the DOM is loaded. It enables all further data flows to be successfully intercepted and logged by the crawler, providing details about the status of the application on page load. The injected script communicate to the background thread of the extension and pass the logged information, which it further relays to the crawler database.

5.2 Exploit Generation

We step into the next phase of our framework to determine whether the existing data flow within an application could be further abused. After we gather data as described in the previous section, we analyze them to understand the nature of data that persists in these storage entities across applications. It provides an overview to help generate accurate exploit data values for each of their original counterparts. We discuss the exploit generation for each domain as follows.

```
var cartData = localStorage.getItem("cart");
// cartData = '{"cartData":{"productId":"PID838", "quantity":3,
↪   "isAvailable":true, "userId":"d033c8be-627d-4cf1-a5f7-859c7d1395dc"},
↪   "timestamp":1589115600000}'

sendCartForCheckout(cartData);
```

Listing 5.1: An example of original data stored at the localStorage

We observe that while cookies mostly store string data, localStorage contains more complex data, usually JSON objects. Similarly, IndexedDB also most often store large and complex JSON objects. The data stored in all these three storage entities are in the form of key-value pairs. Additionally, these data values are often percent-encoded, base64-encoded or sometimes both. We consider these properties while generating and verifying exploit candidates for each of these values.

We parse each of the storage key and value to determine the datatype and generate its exploit counterpart as follows:

1. We first check the string value for it to be base64-encoded, and then decode it, if applicable.
2. We then unescape the data, in case it is per cent-encoded.
3. Further, we determine the datatype of the above-processed value. In case it is a JSON object, we recursively similarly parse each key-value pair.
4. Lastly, for each of the parsed key-value pair, we retain the key as is and replace the original value with an arbitrarily generated string of random length. We generate candidates for each value after identifying the data type and later convert it to the string to preserve the original data type. For instance, the exploit generator creates an integer value to replace the original value with datatype as an integer. We demonstrate this with the help of an example in Listing 5.1 & 5.2.

After we generate exploit candidates for each of the storage entry, we use the chrome extension earlier used for crawling and collecting data to now insert the exploit candidates for each key in respective storage entities at the client. By doing so, we intend to simulate the malicious manipulation of the stored data by an adversary. If the injected exploit candidates into the client-storage further flow to

```
var cartData = localStorage.getItem("cart");
// cartData = '{"cartData":{"productId":"a66AJ8j7vcMOYbT",
↪  "quantity":1451428317,"isAvailable":"3kYmc",
↪  "userId":"0f166328-e6c4-4a28-8c13-4739a43d7099"},
↪  "timestamp":14723957644821}'

sendCartForCheckout(cartData);
```

Listing 5.2: An Example of exploit candidates generated from the original data

any of the sinks, i.e. XMLHttpRequests or PostMessages, we claim that the data flow is vulnerable.

5.3 Exploit Verification

In the last phase of the framework, we intend to verify that an adversary could exploit a target domain, given that there exists a potentially vulnerable data flow between the client and the server. The prerequisite for the framework to do so is to first, identify whether the exploit candidates that it generates and further inject to the client storage persists across multiple sessions or not. The second requirement would be to track the flow of these candidates to the server on subsequent visits by the user. The exploit generation phase provides with a tuple of information which contains the target domain URL with vulnerable data flow, the storage entity, the key and the value which flows from the client to the server.

To verify the persistence of the exploit at the client and its data flow on subsequent visits, we perform the following steps in this phase:

1. We re-run our crawler extension on all the shortlisted domains obtained from the exploit generation phase.

2. For each domain, the crawler records the original data flow until a specified interval of time after page load, as discussed before.

3. The extension then communicates to the exploit server to generate exploits for the recorded data. After producing the exploit candidates, the server sends it back to the extension.

4. Upon receiving the exploit data, the extension injects the generated values based on the key, into their respective storage containers.

5. After it injects all the data, the extension reloads the page and waits for the same amount of time as in Step 2 to log new data flow after the page reloads.

6. Lastly, the extension then triggers the exploit server to validate the data flow. The server now retrieves new requests and postmessages after the page reloads and verifies for the existence of exploit candidates previously generated in Step 3. When it finds such flows, it records into the database in the form of a tuple which contains information as *{targetDomain, sourceType, sinkType, sourceId, sinkId}* for further manual verification, if and when needed.

Implementation Environment

We briefly summarize the implementation environment of our framework as follows. We use headless chromium as our browsing environment automated by the nodejs framework, called *puppeteer* [73]. We instantiate an individual session to investigate each domain separately. The crawler persists on the page for 6 seconds after the page is completely loaded. It additionally waits for 30 seconds in case the page navigates further from the input URL. The crawling extension logs all the data flow that occurs until the browser closes and the nodejs framework intercepts all the XMLHttpRequests issued by the client. All the intercepted data is sent to our exploit server for further analysis, exploit generation and verification.

Chapter 6

Evaluation

In this chapter, we evaluate the web applications on the Internet for the existence of the Client-Side CSRFvulnerability. To this end, we use our proposed detection framework, discussed in Chapter 5 to crawl over the top 1 Million distinct domains and analyze the data flows. We only visit the starting page of these domains in this work. However, one may easily extend the crawling architecture to visit the subdomains and other embedded links, until a predefined depth, n. Since the starting page itself may contain frames pointing to subdomains or any other cross-origin application, we also gather their interaction data. We scrutinize the data accumulated from the first visit to these sites to narrow down the list of domains which exhibit vulnerable data flow. We highlight the observation from this analysis and the dataset chosen for this study in the following sections.

6.1 Evaluation Dataset

To test our framework and further examine the prevalence of this vulnerability, we select the top 1 million domains listed by Tranco as on 02nd February 2020 for evaluation, and run our detection mechanism on them. We crawl over these domain and gather data between 15th - 20th March. We choose recently proposed ranking mechanism by Tranco, as it is well affirmed by the research community and claims to provide comparatively stable domains ranking based on traffic details obtained from four different accepted domain measurement repositories [74].

6.2 Test Application Framework

Before presenting the observations from the analysis on the crawled data, we outline our dummy test application specifically devised to demonstrate the Client-Side CSRF. We build this application to verify and validate the crawling framework, and the resultant vulnerable flows from the analysis. This test application aims to illustrate the innocuous data flow that may occur within the application to perform application-specific or user-specific functionalities between the sources and the sinks considered for this work. An adversary may later abuse these harmless flows to exploit the target domain.

The parent application contains an embedded iframe, which, in turn, consist of another iframe. Though hosted on different domains, these windows have identical functionalities. For each of the sink, i.e., XMLHttpRequest & PostMessages, we craft individual flows from each of the sources which are outline as follows:

1. The application contains skin mode functionality based on user preference as *dark_skin: {true, false}*. Once selected by the user, the value persists across session in the form of a persistent cookie. On each page load, this value flows to the server in the form of additional base64-encoded request header, other than in the default cookie headers, to render the application in selected skin.
2. The other example feature is language preference option. The client stores this value into the localStorage persistent across each page load. The language preference data could be in the form as *lang: {'en', 'de', 'es', 'fr', 'ca'}* This value flows to the server as a base64-encoded and percent-encoded URL parameter in the request-target URL to render the application in the selected language.
3. The sample cart functionality demonstrates the the IndexedDB flow to the XHRs. The cart data is in the form of JSON document, as shown in Listing 5.1. Once set by the application, this value flows to the server as base64-encoded request body, on each page visit, and further shown to the user in an alert window when prompted.
4. We craft similar individual flows for PostMessages. The data flow from one window to another contains secret encoded data along with the message. The target window which receives the message uses this secret to validate

the authenticity of the sender. The parent window communicates with its embedded frame by sending its secret key stored in cookies. This embedded frame communicates with its parent and child window by tagging along with the covert data stored in localStorage for that origin. Similarly, the child frame communicates with its parent and top window by sending secret stored in the IndexedDB for its domain.

5. Each of the target windows also keeps a copy of the secret of other windows for verification. In case the attestation fails, the target window triggers fail-safe defaults pre-defined by the window.

In this manner, we model the real-world functionalities to demonstrate the benign data flows in the test application, susceptible to Client-Side CSRF. We further confirms that these data flows are often found and could also be abused in the real-world applications in later parts of this thesis.

6.3 Vulnerable Data Flows in the Wild

We examine the data collected by the crawler framework and identify critical data flows for each domain. More precisely, we confirm the existence of all the values within the XHRs and the PostMessages for which the crawler had previously registered a *GET* operation in either of the source storage containers, before the request/postmessage was issued. We look for these values in sink targets such as to request header and body, and in the message data in PostMessages. We further examine the request and the postmessage target URLs and their parameters, for the completeness of our analysis. We also consider base64-encoded and percent-encoded data when comparing the source and sink values and further decode them to their base form before comparison for accurate results.

We report the result obtained from the above analysis in Table 6.1. The figures in this table represent distinct domains which exhibit dangerous flows categorized into their sources and sinks, as rows and columns, respectively. Also, for each category, the numbers indicate the existence of the data flow from a particular source to its corresponding sink, independent of other identified source to sink flows within the same domain. We note that a single application may contain multiple data flows originating from the same or distinct sources and similarly, ending up at

Storage Source	**XHR Sink**	**PostMessage Sink**	**Total Domains**
Cookies	73,089	18,744	77,019
LocalStorage	75,312	43,024	81,425
IndexedDB	13,566	3,401	13,657
Total Domains	132,068	54,949	139,021

Table 6.1: An overview of the distinct number of domains that contain at least one data flow detected between the sources (columns) and the sinks (rows) from crawling on Top 1 Million Sites.

same or different sinks. For instance, there may exist only one data flow between a cookie and a XHR, whereas, there may exist multiple data flows between a single IndexedDB record flowing to both XHR and the PostMessage. Therefore, the gross flows from a particular source, and similarly, the aggregate flows to each of the sinks does not correspond to the total of all distinct domains (in the bottom-right cell) as there may exist multiple flows for each domain arising from and ending-up at different sources and sink, respectively. We consider those domains relevant for our study for which we identify at least one vulnerable source to sink flow.

We observe that over 13.9% of the top 1 million domains contain data flows which could be abused by an adversary in case it carries sensitive information to perform a state-changing operation at the server-side. The figures suggest that both the cookies and the localStorage are extensively used by a significant number of domains to store data spanning across sessions and further communicated to the server on every visit. On the contrary, we note that not many applications use IndexedDB for the same purpose. We believe that this is due to the complexity of the IndexedDB APIs when compared to the other storage APIs as well as the nature of the data that the application stores at the client. Moreover, since IndexedDB is also accessible by the service workers set for that origin, the communications alternatively may occur in the background through these workers [75].

We note that while cookie data mostly flow to the XHRs, localStorage is a common source to form postmessages when compared to the cookies, as indicated by the figures. Further, it may appear that XHRs is more widely used than the postmessage API to communicate stateful data to the server. However, this is not true as we analyze requests of type *xhr* from a total of 439,627 distinct domains among which 30.04% of those domains carry stateful data to the server.

Storage Source	**XHR Sink**	**PostMessage Sink**	**Total Flows**
	Total (Encoded %)	Total (Encoded %)	Total (Encoded %)
Cookies	21,076,348 (5.78%)	6,748,795 (0.82%)	27,825,143 (4.70%)
LocalStorage	1,263,344 (1.68%)	487,176 (0.24%)	1,750,520 (1.28%)
IndexedDB	1,937,436 (–)	163,531,463 (–)	165,468,899 (–)
Total Flows	24,277,128 (5.10%)	170,767,434 (–)	195,044,562 (0.66%)

Table 6.2: Total data flows detected between the identified sources(columns) and sinks(rows) over domains in 6.1. We additionally report the encoded data flows (in %) detected as relative to the total flows for each *{source, sink}* combination.

On the other hand, we collect the postmessages from a total of 162,986 distinct domains and observe that over 33.72% of these domains carry potentially sensitive information to the server. Upon analysis, we determine that the integration of third-party resources, *iframes*, in particular, is responsible for the high rate of data flow detected.

We further report the detailed view of each data flow between individual storage sources and sink for all the domains in Table 6.2. We identify the data flows which are encoded before it is communicated by the sink to the server and present them together. We consider these encoding schemes for this study as it signifies that the application developer uses some client-side mechanism, such as *encodeURIComponent* and *btoa*, to abstract data from unwanted manipulation. Nevertheless, we note that it does not indicate that there exists any useful integrity protection or validation mechanism for these data in place at the client.

Contrary to the number of domains which carry data from the IndexedDB to either of the sinks, we observe that over 84.83% flows that we have collected, originates from the IndexedDB. We believe that this large fraction of flows exist as it can store large and complex data which can be used by the web applications that perform heavy processing at the client-side on every visit. Additionally, we note that only a negligible number of data flow showed any form of encoding. The same is also true for localStorage, whereas cookie data are occasionally encoded. It shows implicit trust in the client-side storage by the application developers, particularly on the localStorage and the IndexedDB. The results are in line with other previous works focused on the integrity of the web-storage discussed in 2.1.

On the other hand, we observe that over 87.55% of the data flow that originates from any of the identified sources end up at the PostMessage sink. This follows our observation as stated above that website which integrates a large number of third-party resources store more data at the client and further send them to their respective server, using the PostMessage API. It is important to note that over 86.8% of the data that flows to XHRs originate from cookies. We further investigate them and identify that these data flows are often used as the parameters of the target URL, additionally carrying stateful information.

We shortlist a total of 139,021 domains from the initial phase of analysis and classify them as potentially vulnerable to Client-Side CSRF. Further, we examine these shortlisted domains to confirm the exploitability in the later stage of this study.

Chapter 7

Results

In the previous chapter, we used our proposed data collection method to detect and evaluate the web applications which are vulnerable to **clientcsrf!**. We shortlisted 139,021 websites which demonstrated these susceptible flows an attacker could potentially exploit. To verify the exploitability, we re-evaluate these domains in this chapter by utilizing our exploit generation and verification method and present the obtained results. We then highlight the patterns observed among these flows, that may be responsible for the existence of these vulnerabilities on the web. In the end, we discuss particular instances where we find these flows among web applications.

7.1 Exploitable Domains with Persistent Data Flow

To validate the vulnerability of the shortlisted domains identified in the previous phase, we re-run our crawler framework on these domains. The exploit server generates suitable candidates and forwards it to the crawler to insert them on-the-fly into the client-storage of these domains. The crawler then reloads the website to confirm the persistence of injected data, as also explained in Section 5.2 and 5.3. Lastly, we collect all the XHRs and the PostMessages after the page reloads and analyze them to determine the data flow, which contains the exploit candidates injected by the framework. We report the results obtained after the verification of these domains in Table 7.1 as follows.

Storage Source	XHR Sink	PostMessage Sink	Total Domains
Cookies	16,258	1,857	17,705
LocalStorage	13,210	605	13,684
IndexedDB	11	9	15
Total Domains	28,099	2,385	29,815

Table 7.1: Distinct number of domains which contain at least one exploitable data flow verified by our exploit framework, among 139,021 domains identified as in Table 6.1

We report that over 21.44% of the shortlisted domains or a total of 2.98% among the top 1 million web applications are susceptible to the **clientcsrf!** attack. As with Table 6.1, the aggregate flows from specific sources and sinks do not correspond to the sum of all distinct domains as the data flows overlap for these domains in many cases. These domains contain at least one data flow originating from the client-storage further used by either the XHR or the PostMessage API to communicate to the server, as verified by our automated framework. It is important to note that this figure represents a strict lower bound of the number of exploitable domains as we only investigate those data flows in this work that occurs on the top-level page of the web application. Further, we only consider those values for our analysis whose length is greater than one character, to avoid false positives. Nevertheless, we understand that these storage values may also often carry crucial stateful information to the application. We manually review a total of 200 randomly chosen data flows to validate the findings by our exploit framework and observe no false positives.

We infer that since the injected exploit candidates persist across the session and the application uses these exploit data to perform operations at the server, there is no effective integrity-check or data validation strategy used at the client-side in these applications. We note similar trends of data flow in the exploit validation stage, as also observed initially in Table 6.1, for the sources and the sink. The cookies and localStorage are the popular choices for the application developers to store data at the client. However, we suspect that the reason behind the lower number of data flows originating from the IndexedDB is because of a validation mechanism used for the data that originates from this container.

Similarly, a majority of these domains use XHRs to send this data to the server

Storage Source	Injection Point	Cookies	LocalStorage	IndexedDB
XHR	Header	859	132	–
	Body	6,349	2,463	–
	Target URL	10,335	10,902	11
PostMessages	Message	1,527	571	–
	Target URL	333	36	9
Total Domains	-	17,705	13,684	15

Table 7.2: Identified usage or injection point of the data at the sink flowing from the clients' persistent storage for verified exploitable domains in 7.1

when compared to the PostMessage sink. Based on the number of flows to the data sink, it is interesting to note that the application takes extra caution to form the PostMessages in contrary to the XHRs, and may demand further investigation. We highlight that, as observed in Table 6.2, a majority of the data flows are not found to be encoded, and thus, we skip to examine this behavior in this stage of analysis.

7.2 Data Flow Patterns

To develop a further understanding of the stateful data stored at the client as well as its usage by the application, we further investigate the injection point of these persistent data into the sinks. For instance, data that originates from the localStorage may define properties in the request body while it forms the URL parameters of the PostMessage target. We identify these data flows for each of the exploitable domains and report them in Table 7.2. This table categorizes different injection points available at the sink, i.e., XHR and the PostMessage (as rows), for each of the data sources (as columns).

We see that most of the data flow end up as URL parameters into either of the sinks. By further investigating these flows, we find that these are often the session or user identifiers set by the target server or the third-party domains. Besides, they carry other forms of data such as language preferences, cookie consent & policies and other application-wide configurations, stored in the client. When communication occurs with any third-party domains, we see *api keys* as one of the most common parameters to flow from the cookies. In case of XHRs, the flows to request headers often consist of the *x-xsrf-tokens* for authentication at the server. These tokens flow from either localStorage or the cookies and do not change throughout the session in most cases. The application often sends tracking data to the third-party domain using XHRs by putting data into the request body for which there does not exist any need to receive the response.

We observe similar trends for the data flow to PostMessages injection point as most of them end up to form the target URL parameters. These are most often tracking identifiers, third-party session identifiers, the parent domain or the referrer who sends these request and other user preference data. Further, the message contains analytics data such as the user browsing history on the parent domain, the actions performed, and so on. In most cases, we observe that the websites typically use the PostMessages API to send data to third-party domains that provide analytics or tracking services. While, in case of XHRs, the request targets are often the subdomains or third-party domains which provide services such as chatbots and password-management services.

7.3 Case Study

Though we do not find any websites where a vulnerable data flow results in the execution of security-critical functionality at the server in the context of an authenticated session, we report few generic cases specific to the state of the applications. We notice that there are certain third-party domains which cater services and resources to several mainstream web applications. The web applications communicate with these service providers via XHR or PostMessage and send additional information stored at the persistent APIs in the form of URL parameters. This information contains information such as tracking data, user interactions, application configuration and cookie consents. By influencing the data stored at the client, an adversary can force the application to cause unintended side-effects such as disabling chat windows, allowing trackers and third-party cookies without actual consent from the user and fake analytics profile. This way, a shared service provider which uses these data flows to render services to multiple applications may also expose the parent applications to this threat.

In a few applications, we observe that manipulating stored data by injecting arbitrary content leads to side-effects such as crash or failure to load the requested data for as long as the data persists in the storage. We verify this issue by observing the side-effects on applications while performing manual analysis on 200 randomly selected domains from our dataset.

Chapter 8

Discussion

From our study on the current state of web applications and their interactions, we understand that the persistent storage APIs are reasonably common among these applications and are widely used by the developers to store data and further utilize them to across sessions as well as origins. In this chapter, we discuss a few additional characteristics of the data sources and sinks that we observe in the wild and review generic data validation mechanisms that could be used by the application developers. We also highlight the limitations of our study to define the boundaries of our proposed framework and the reported results. We then suggest the future avenues of research on this threat which could be interesting for fine-grained analysis and to discover effective countermeasures against this attack.

8.1 Varied Behavior of Sources & Sinks

The results obtained from initial analysis from the top 1 million domains (as listed in Table 6.2) suggests that even though the client sends data to the server stored at these persistent data stores, we observe only a few instances where the application developer sanitize this data by using encoding schemes before re-using it. While these encoding schemes defend against several web attacks such as XSS and remote file inclusion, they do not specify anything about the integrity of the data. Thus, we do not consider them as protective measures against this vulnerability. As also indicated in previous works, the server could validate the integrity of the data it receives, if the data was created and stored by the server previously. However, in the current scenario, there are instances where the client generates sensitive

Data Sinks	**Data Flows**	**Distinct Domains**
XHR Sink	5,897,552	230,222
PostMessage Sink	99,726,111	15,522
Total	105,623,663	232,205

Table 8.1: Data flows identified between the URL Parameters and the sinks. We also report the distinct domains where these flows are found.

data at runtime and store them in these persistent data stores. Moreover, with heightened integration of third-party resources, we observe that this trend has increased substantially such that the client generates stateful data for the current session and often send them to these third-party domains for various purposes.

Apart from the data sources that we have discussed in detail so far, i.e., cookies, localStorage and the IndexedDB, we also highlight that the URL parameters of the top-level page may carry sensitive information about the user or application and further flow to either of the sinks, directly or indirectly via persistent data stores. Though we do not verify the exploitability of the data flows that originate from these URL parameters, we analyze these flows over the top 1 million domain and report the results obtained in Table 8.1. We observe that over 23.2% of these websites have at least one potentially vulnerable flow to the sinks. Upon investigating a few of these flows manually, we observe that these are often URL fragments which flow to the third-party domains to indicate the origin of the message along with other information such as session identifiers, API keys and user profile for analytics.

We observe a significantly lower number of flows originating from the IndexedDB when compared to cookies and the localStorage in contrast to the fact that they often contain a humongous amount of data. We further investigate this with the help of our crawler and find that among 24,871 distinct websites which use the IndexedDB to store data, 1,850 of them also register service workers for that origin. We further discover that among these origins with service workers, 1,234 of them do not contain any data flow to either XHRs or the PostMessages that originates from the IndexedDB. Thus, we believe the applications communicate to this datastore using the service workers and not through the sinks that we consider for this work.

8.2 Limitations

In this work, we propose an automated framework to detect and verify the existence of Client-Side CSRFvulnerabilities on the web at a large scale. We understand that while our proposed framework successfully identifies the source to sink flows within applications that are relevant to this study as well as verify the associated threat for any domain, there are certain limitations with our approach which we discuss as follows.

As defined earlier in Section 3.1.2, the state in the context of our study may either refer to application-specific or user-specific state at the client. Since we only crawl on the top-level page of each of the domains without actually logging-in to the application, we do not observe those data flow that contains data specific to the user-state at the client. We believe that this would require manual intervention in addition to the automated crawling and verification framework to log-in to the application and perform security-sensitive operations. We further highlight that our crawler only visits the top-level page for a given domain and not any of its subdomain or embedded links. Nonetheless, this could be achieved by our framework by simply extending it to visit all of the embedded links, until a predefined depth n.

8.3 Future Work

Based on our current approach to detect Client-Side CSRF, we suggest the future line of work that could further help deepen our knowledge of this exploit surface and provide information for the application developers to mitigate against such threats on the web. An examination of the data flows which carry persistent user-state at the client behind the login is essential to understand the full landscape of the menace, as we believe that exploitation against user-state would have more severe consequences than the application-state. While it does not apply to single-page applications, an analysis on the selected list of domains could be performed in semi-automated fashion behind the login.

As we saw in Section 8.1, there are data flows which originate from the URL parameters or the links embedded on the current page which further flows to either the client-storage or directly to the data sink. Since we do not examine the exploitability of such flows in this study, a further investigation of these flows to determine the vulnerability of the source data and its insecure usage is essential. Additionally, some domains integrates multiple third-party domains where each of these domains generate sensitive data for the user and communicate them across the origin using PostMessages. In this case, the domain that receives this data generated by the third-party acts as the server and has no mechanism to validate the integrity of the message received. Thus, a third-party domain which could be under the influence of an attacker could further serve as the data source for such vulnerable flows.

We also suspect that similar precarious data flow exists which may originate from the IndexedDB or other persistent storage APIs and used by the service workers for that origin. Thus, an explicit review of these data flow would make sense to invalidate this threat. At last, we note that an explicit mapping of the verified exploitable domains to the attacker model discussed in this work is due and could be addressed in the follow up work.

Chapter 9

Conclusion

Cross-Site Request Forgery (CSRF) is perceived as one of the most powerful web attacks and is broadly addressed in the past by security researchers. After being counted among one of the top 10 web application threats for over a decade, we observe that these threats have reduced over time due to various mitigation strategies proposed by the researchers. In this study, we describe a new exploit surface on the web application that an attacker may abuse to carry out CSRF even when the application uses appropriate countermeasures to defend against CSRF, called Client-Side CSRF. To our knowledge, this window of vulnerability has not yet been addressed by the scientific community which arises due to the insecure usage of the client-side state, stored in one of the persistent storage containers, to perform sensitive operations at the server.

In this work, we conduct a first empirical study over the top 1 million applications to detect this peril on the web. Based on the observation, we state that over 2.9% of all these domains are susceptible to the underlined threat and could be exploited by an adversary who can influence the data stored at the client. When concentrating only on those domains that contain data flows originating from the persistent storage and further sent to the server via either of the sink APIs, i.e., XMLHttpRequests and PostMessages, we observe that 21.8% of all them are vulnerable to this attack. To investigate these flows, we propose an automated framework that simulates an attacker by generating exploit candidates as well as verifying their persistence and its further flow to the server. We further outline the threat model that describes the attackers' capacity to influence the data stored at the client, leading to exploitation.

We infer from the obtained results that the application developers inherently trust on the integrity of these persistent data stores at the client as also outlined by previous works and is used by the application without any integrity-check, encoding mechanism or validation in many cases. We discuss the flow patterns observed in the wild to identify the nature of these data flows as well as its benign usage within the application while, at the same time, show that when used without any validation mechanism, it could lead to persistent threat arising from the client-side of the application. We further discuss specific cases to stress the severity of this issue across domains. While CSRF is considered to be a server-side flaw in which the server is unable to distinguish between an original and a fake request, our study reveals that this vulnerability stems from the client-side. The substantial number of domains reported from our study suggests that the underlined threat is indeed an issue on the web and could be further abused by an adversary if the application stores data to re-use them later without taking additional consideration.

Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
AJAX	Asynchronous JavaScript & XML
CORS	Cross-Origin Resource Sharing
CSP	Content-Security Policy
CSRF	Cross-Site Request Forgery
DOM	Document-Object Model
FQDN	Fully Qualified Domain Name
HSTS	HTTP Strict-Transport Security
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
JS	JavaScript
JSON	JavaScript Object Notation
MitB	Man-in-the-Browser
MitM	Man-in-the-Middle
ML	Machine Learning
OWASP	Open Web Applications Security Group

PWA	Progressive Web Applications
RDBMS	Relational Database Management System
SOC	Same-Origin Communication
SOP	Same-Origin Policy
SQL	Structured Query Language
SQLi	SQL Injection
TLS	Transport-Layer Security
URL	Uniform Resource Locator
WWW	World Wide Web
W3C	World Wide Web Consortium
WHATWG	Web Hypertext Application Technology Working Group
XHR	XMLHttpRequest
XML	Extensible Markup Language
XSS	Cross-Site Scripting
XSRF	Cross-Site Request Forgery
XSSI	Cross-Site Script Inclusion

List of Figures

List of Tables

Bibliography

[1] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen, "Webjail: least-privilege integration of third-party components in web mashups," in *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011, pp. 307–316.

[2] L. Xing, Y. Chen, X. Wang, and S. Chen, "Integuard: Toward automatic protection of third-party web service integrations." in *NDSS*, 2013.

[3] L. Cui, S. Huang, F. Wei, C. Tan, C. Duan, and M. Zhou, "Superagent: A customer service chatbot for e-commerce websites," in *Proceedings of ACL 2017, System Demonstrations*, 2017, pp. 97–102.

[4] B. Stock, M. Johns, M. Steffens, and M. Backes, "How the web tangled itself: Uncovering the history of client-side web (in) security," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 971–987.

[5] S. Calzavara, R. Focardi, M. Squarcina, and M. Tempesta, "Surviving the web: A journey into web session security," *ACM Comput. Surv.*, vol. 50, no. 1, Mar. 2017. [Online]. Available: https://doi.org/10.1145/3038923

[6] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan, "Navex: Precise and scalable exploit generation for dynamic web applications," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 377–392.

[7] C. Cetin, D. Goldgof, and J. Ligatti, "Sql-identifier injection attacks," in *2019 IEEE Conference on Communications and Network Security (CNS)*, 2019, pp. 151–159.

[8] S. Lekies, B. Stock, and M. Johns, "25 million flows later: Large-scale detection of dom-based xss," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 1193–1204.

[9] B. Stock, S. Pfistner, B. Kaiser, S. Lekies, and M. Johns, “From facepalm to brain bender: Exploring client-side cross-site scripting,” in *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, 2015, pp. 1419–1430.

[10] M. Balduzzi, C. T. Gimenez, D. Balzarotti, and E. Kirda, “Automated discovery of parameter pollution vulnerabilities in web applications.” in *NDSS*. Citeseer, 2011.

[11] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, “Efficient and flexible discovery of php application vulnerabilities,” in *2017 IEEE european symposium on security and privacy (EuroS&P)*. IEEE, 2017, pp. 334–349.

[12] A. Sudhodanan, R. Carbone, L. Compagna, N. Dolgin, A. Armando, and U. Morelli, “Large-scale analysis detection of authentication cross-site request forgeries,” in *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, 2017, pp. 350–365.

[13] S. Lekies, B. Stock, M. Wentzel, and M. Johns, “The unexpected dangers of dynamic javascript,” in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 723–735.

[14] BBC. (2020) Clearview AI: Face-collecting company database hacked. [Online]. Available: https://www.bbc.com/news/technology-51658111

[15] M. Weissbacher, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, “Zigzag: Automatically hardening web applications against client-side validation vulnerabilities,” in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 737–752. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/weissbacher

[16] P. Saxena, S. Hanna, P. Poosankam, and D. Song, “Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications.” in *NDSS*, 2010.

[17] L. K. Shar and H. B. K. Tan, “Mining input sanitization patterns for predicting sql injection and cross site scripting vulnerabilities,” in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 1293–1296.

[18] S. Lekies and M. Johns, “Lightweight integrity protection for web storage-driven content caching,” in *6th Workshop on Web*, vol. 2, 2012.

[19] S. Hanna, R. Shin, D. Akhawe, A. Boehm, P. Saxena, and D. Song, “The emperor’s new apis: On the (in) secure usage of new client-side primitives,” in *Proceedings of the Web*, vol. 2, 2010.

[20] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver, “Cookies lack integrity: Real-world implications,” in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 707–721. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/zheng

[21] S. Sivakorn, A. D. Keromytis, and J. Polakis, “That’s the way the cookie crumbles: Evaluating https enforcing mechanisms,” in *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society*, ser. WPES ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 71–81. [Online]. Available: https://doi.org/10.1145/2994620.2994638

[22] M. Steffens, C. Rossow, M. Johns, and B. Stock, “Dont́ Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild.” 2019.

[23] CVE Details. Vulnerability Details : CVE-2002-1648. [Online]. Available: https://www.cvedetails.com/cve/CVE-2002-1648/

[24] N. Jovanovic, E. Kirda, and C. Kruegel, “Preventing cross site request forgery attacks,” in *2006 Securecomm and Workshops*. IEEE, 2006, pp. 1–10.

[25] A. Barth, C. Jackson, and J. C. Mitchell, “Robust defenses for cross-site request forgery,” in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 75–88.

[26] T. Alexenko, M. Jenne, S. D. Roy, and W. Zeng, “Cross-site request forgery: Attack and defense,” in *2010 7th IEEE Consumer Communications and Networking Conference*, 2010, pp. 1–2.

[27] P. De Ryck, L. Desmet, W. Joosen, and F. Piessens, “Automatic and precise client-side protection against csrf attacks,” in *European Symposium on Research in Computer Security*. Springer, 2011, pp. 100–116.

[28] R. Pelizzi and R. Sekar, “A server- and browser-transparent csrf defense for web 2.0 applications,” in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 257–266. [Online]. Available: https://doi.org/10.1145/2076732.2076768

[29] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow, “Deemon: Detecting CSRF with dynamic analysis and property graphs,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1757–1771.

[30] S. Calzavara, M. Conti, R. Focardi, A. Rabitti, and G. Tolomei, “Mitch: A machine learning approach to the black-box detection of CSRF vulnerabilities,” in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 528–543.

[31] J. Dahse and T. Holz, “Static detection of second-order vulnerabilities in web applications,” in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 989–1003. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/dahse

[32] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, “Precise client-side protection against dom-based cross-site scripting,” in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 655–670.

[33] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, “You are what you include: large-scale evaluation of remote javascript inclusions,” in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 736–747.

[34] M. Ikram, R. Masood, G. Tyson, M. A. Kaafar, N. Loizon, and R. Ensafi, “The chain of implicit trust: An analysis of the web third-party resources loading,” in *The World Wide Web Conference*, ser. WWW ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2851–2857. [Online]. Available: https://doi.org/10.1145/3308558.3313521

[35] S. Son and V. Shmatikov, “The postman always rings twice: Attacking and defending postmessage in html5 websites.” in *NDSS*, 2013.

[36] C. Guan, K. Sun, Z. Wang, and W. Zhu, "Privacy breach by exploiting postmessage in html5: Identification, evaluation, and countermeasure," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 629–640.

[37] A. Barth, C. Jackson, and J. C. Mitchell, "Securing frame communication in browsers," *Communications of the ACM*, vol. 52, no. 6, pp. 83–91, 2009.

[38] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz, "The web never forgets: Persistent tracking mechanisms in the wild," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 674–689. [Online]. Available: https://doi.org/10.1145/2660267.2660347

[39] A. Sudhodanan, S. Khodayari, and J. Caballero, "Cross-origin state inference (cosi) attacks: Leaking web site states through xs-leaks," *arXiv preprint arXiv:1908.02204*, 2019.

[40] W3C. (1999) Hypertext Transfer Protocol – HTTP/1.1. [Online]. Available: https://tools.ietf.org/html/rfc2616

[41] ——. (2011) HTTP State Management Mechanism. [Online]. Available: https://tools.ietf.org/html/rfc6265

[42] D. M. Kristol, "Http cookies: Standards, privacy, and politics," *ACM Trans. Internet Technol.*, vol. 1, no. 2, p. 151–198, Nov. 2001. [Online]. Available: https://doi.org/10.1145/502152.502153

[43] I. Dacosta, S. Chakradeo, M. Ahamad, and P. Traynor, "One-time cookies: Preventing session hijacking attacks with stateless authentication tokens," *ACM Trans. Internet Technol.*, vol. 12, no. 1, Jul. 2012. [Online]. Available: https://doi.org/10.1145/2220352.2220353

[44] G. Franken, T. Van Goethem, and W. Joosen, "Who left open the cookie jar? a comprehensive evaluation of third-party cookie policies," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 151–168.

[45] A. Lerner, A. K. Simpson, T. Kohno, and F. Roesner, "Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016," in *25th USENIX Security Symposium*

(USENIX Security 16). Austin, TX: USENIX Association, Aug. 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lerner

[46] W3C. (2016) Web Storage. [Online]. Available: https://www.w3.org/TR/2016/REC-webstorage-20160419/

[47] G. Anthes, "Html5 leads a web revolution," *Commun. ACM*, vol. 55, no. 7, p. 16–17, Jul. 2012. [Online]. Available: https://doi.org/10.1145/2209249.2209256

[48] Mozilla Developer Network. (2019) IndexedDB. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API

[49] E. Yang, D. Stefan, J. Mitchell, D. Mazières, P. Marchenko, and B. Karp, "Toward principled browser security," in *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, 2013.

[50] D. Akhawe, P. Saxena, and D. Song, "Privilege separation in html5 applications," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12).* Bellevue, WA: USENIX, 2012, pp. 429–444. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/akhawe

[51] J. Weinberger, A. Barth, and D. Song, "Towards client-side html security policies." in *HotSec*, 2011.

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

[53] M. Decat, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen, "Towards building secure web mashups," in *OWASP AppSec Research 2010, Date: 2010/06/23-2010/06/24, Location: Stockholm, Sweden*, 2010.

[54] WHATWG. (2020) Web-Messaging. [Online]. Available: https://html.spec.whatwg.org/multipage/web-messaging.html

[55] Z. Kessin, *Programming HTML5 applications: building powerful cross-platform environments in JavaScript.* " O'Reilly Media, Inc.", 2011.

[56] J. Hodges, C. Jackson, and A. Barth, "Http strict transport security (hsts)," *URL: http://tools. ietf. org/html/draft-ietf-websec-strict-transport-sec-04*, 2012.

[57] Chromium Project. HSTS Preload List. [Online]. Available: https://hstspreload.org/

[58] IETF. (2012) HTTP Strict Transport Security (HSTS). [Online]. Available: https://tools.ietf.org/html/rfc6797

[59] K. Krombholz, W. Mayer, M. Schmiedecker, and E. Weippl, ""i have no idea what i'm doing" - on the usability of deploying HTTPS," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1339–1356. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/krombholz

[60] M. Kranch and J. Bonneau, "Upgrading https in mid-air," in *Proceedings of thz 2015 Network and Distributed System Security Symposium. NDSS*, 2015.

[61] K. Krombholz, H. Hobel, M. Huber, and E. Weippl, "Advanced social engineering attacks," *Journal of Information Security and applications*, vol. 22, pp. 113–122, 2015.

[62] D. Akhawe, W. He, Z. Li, R. Moazzezi, and D. Song, "Clickjacking revisited: A perceptual view of UI security," in *8th USENIX Workshop on Offensive Technologies (WOOT 14)*. San Diego, CA: USENIX Association, Aug. 2014. [Online]. Available: https://www.usenix.org/conference/woot14/workshop-program/presentation/akhawe

[63] R. Heartfield and G. Loukas, "A taxonomy of attacks and a survey of defence mechanisms for semantic social engineering attacks," *ACM Comput. Surv.*, vol. 48, no. 3, Dec. 2015. [Online]. Available: https://doi.org/10.1145/2835375

[64] M. Simeonovski, G. Pellegrino, C. Rossow, and M. Backes, "Who controls the internet? analyzing global threats using property graph traversals," in *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 647–656.

[65] A. Doupé, "History and future of automated vulnerability analysis," in *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '19. New York, NY, USA: Association

for Computing Machinery, 2019, p. 147. [Online]. Available: https://doi.org/10.1145/3322431.3326331

[66] Mozilla Developer Network. Object.defineProperty. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty

[67] ——. (2020) Document.cookie. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie

[68] ——. (2020) Set-Cookie. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie

[69] ——. (2019) Using IndexedDB. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Using_IndexedDB

[70] ——. (2020) HTMLIFrameElement.contentWindow. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/HTMLIFrameElement/contentWindow

[71] ——. (2019) MessageEvent. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/MessageEvent

[72] Chrome Developers Team. Content Scripts. [Online]. Available: https://developer.chrome.com/extensions/content_scripts#declaratively

[73] Google Developers. Puppeteer. [Online]. Available: https://developers.google.com/web/tools/puppeteer

[74] V. L. Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, "Tranco: A research-oriented top sites ranking hardened against manipulation," *arXiv preprint arXiv:1806.01156*, 2018.

[75] W3C. (2019) Service Workers 1. [Online]. Available: https://www.w3.org/TR/service-workers-1/