Are your Sites Truly Isolated? Automatically Detecting Logic Bugs in Site Isolation Implementations

Jan Drescher
TU Braunschweig
jan.drescher@tu-braunschweig.de

David Klein
TU Braunschweig
david.klein@tu-braunschweig.de

Martin Johns
TU Braunschweig
m.johns@tu-braunschweig.de

***Abstract*—Site Isolation is one of the core security mechanisms of a modern browser. By confining aspects such as the JavaScript Just-in-Time compiler or the HTML rendering to a sandboxed process, web browsers significantly reduce the impact of memory corruption errors. In addition, the mechanism protects against microarchitectural attacks such as Spectre. When using Site Isolation, the browser confines all processing related to a site to its own sandboxed process. All communication with the privileged browser process is done via exchanging IPC messages. This, however, requires the browser process to keep track of which renderer processbelongs to which site, as otherwise, an attacker can abuse a memory corruption issue in the renderer to attack other sites by sending malicious IPC messages. This, in turn, would allow attackers to leak sensitive data, such as cookies, or even achieve Universal Cross-Site Scripting.**

This work presents the first automatic approach to detect such vulnerabilities, called Site Isolation bypasses, in Firefox and Chrome. For this, we propose a novel oracle to detect the semantic bugs that cause Site Isolation bypass vulnerabilities by flagging cross-site data leaks on the process level. In addition, we design a fuzzer that simulates a compromised renderer process, trying to use the browser process as a confused deputy by hooking into the IPC communication. Our work uncovered four security vulnerabilities in Chrome and Firefox: three less severe bugs leak data cross-site while the fourth bug facilitates complete control over the victim site.

I. Introduction

Web browsers are a lucrative target for attacks because they have a large user base and process primarily untrustworthy input by design. To reduce the impact of security bugs in handling untrustworthy content, web browsers process it in a low-privileged, sandboxed renderer process, with only the high-privileged browser process having access to the host system. The renderer performs many tasks that are prone to memory bugs, such as HTML parsing and JIT compiling JavaScript code. Kocher et al. [1] discovered the microarchitecture attack Spectre, that allows the leak of all data in the same process. This resulted in a radical restructuring of the browser's security model, culminating in *Site Isolation*.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240902
www.ndss-symposium.org

Site Isolation (SI) is a recent browser security architecture that isolates web applications by *site* in separate, sandboxed renderer processes to mitigate Spectre and renderer compromises [2]. A *site* is defined as the tuple of *scheme* (e.g., `https`) and *extended Top-Level Domain plus one subdomain* (short eTLD+1, e.g., `example.com`). The security of Site Isolation relies on process isolation provided by the operating system. It also relies on the security of the privileged browser process. The browser process communicates with all renderer processes via inter-process communication (IPC) to provide them with cross-site networking and communication capabilities. These capabilities are restricted by the same-origin policy and Cross-Origin Resource Sharing (CORS). The browser process must correctly track the site context of every renderer process and enforce these security policies. Bugs in the site mapping or the policy enforcement lead to Site Isolation bypass vulnerabilities that allow an attacker to execute malicious JavaScript in the context of another site or steal cookies.

Site Isolation was rolled out in 2018 in Chrome [2] and in 2021 in Firefox [3]. At the moment of writing, Safari developers started implementing Site Isolation. Most Site Isolation bypass bugs discovered since then required the attacker to have compromised the sandboxed renderer process to be exploitable. This produced a relatively high barrier. But assuming that the sandbox is secure and sandbox escapes are impossible, Site Isolation bypasses are the most lucrative attack vector to utilize a renderer compromise. The vendors of Chrome and Firefox rank Site Isolation bypass vulnerabilities in the second-highest tier of their security bug bounty programs.

In contrast to memory corruptions that the Address Sanitizer [4] can detect, detecting semantic bugs such as Site Isolation bypass bugs is hard because they do not produce easily visible crashes [5]. To detect these vulnerabilities, we need to infer which process is under the control of the attacker (i.e., processes attacker-provided inputs) and if this process is able to access cross-site data, leading to our first research question:

RQ1 How can SI bypass vulnerabilities (i.e., cross-process data leaks) be reliably detected?

To this end, we propose and evaluate a leak sanitizer as an oracle for Site Isolation bugs. Running the complete browser, our sanitizer detects when a known secret value from the victim site is leaked to the attacker process. We propose a second oracle, the process sanitizer, to detect process-reuse bugs that do not produce visible cross-process leaks but are nevertheless vulnerable to Spectre attacks.

However, it is not sufficient to only detect successful SI bypasses; the bug must be triggered first. The vulnerability lies in the browser process's reaction to IPC messages that a normal renderer process would not send. A compromised renderer can exhibit arbitrary behavior and send malicious IPC messages on all available interfaces. This leads to our second research question:

RQ2 How to model the arbitrary malicious behavior of the compromised renderer?

To answer this question, we systematically analyze past SI bypasses, discovering that most proofs-of-concept for SI bypasses require only minor changes in the behavior of the renderer process: The renderer compromise is most frequently used to circumvent security checks in the renderer or spoof origin-related parameters in IPC messages to privileged processes. Based on this discovery, we propose a fuzzing approach that intercepts and modifies IPC messages to simulate a compromised renderer process, aiming to trigger Site Isolation bypass bugs.

Our contributions are the following:

- We are the first to systematically analyze and classify the 39 known Site Isolation bypass vulnerabilities in Chrome and Firefox. Leveraging insights from this analysis, we identify the necessary preconditions to trigger these bugs.
- We implement a Web IDL-driven fuzzer[1] that triggers diverse and meaningful inter-process communication and simulates a renderer compromise by intercepting and manipulating the IPC messages emitted by the renderer.
- We propose novel oracles for Site Isolation bypasses, that observe the data flows between processes and detect cross-site data leaks and process-reuse.
- We run a month-long fuzzing campaign targeting Chrome and Firefox, discovering four security bugs and reporting them to the developers. The bugs differ in severity: Three less severe bugs leak data cross-site. The fourth bug facilitates complete control of the victim site. It was assigned a CVE, and we were awarded an $8000 bug bounty.

II. Background

In this section, we introduce the Site Isolation architecture and the technologies that SI builds upon. Afterward, we analyze the common causes for Site Isolation bypass vulnerabilities by analyzing previous SI bypass bugs from the Chrome and Firefox browser.

[1]https://github.com/si-bypass-fuzzing

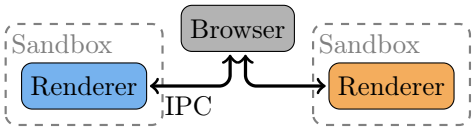

Fig. 1. Site Isolation architecture

A. Site Isolation

To mitigate memory bugs in the HTML parser or the JavaScript engine, which facilitate remote code execution (RCE), modern browsers process potentially malicious HTML and scripts in a separate process inside a sandbox. This process is called *renderer process*, *content process*, or *renderer*. We will use the terms *renderer process* or *renderer* to refer to this class of processes. The sandbox prevents the renderer from accessing the host system, thus mitigating the impact of RCE vulnerabilities in the renderer. Only the privileged *browser process* or *parent process* has access to the host system. We will use the term *browser process* for this process. The browser process and the renderer communicate via inter-process communication (IPC). The browser process interacts with the user, fetches documents via the network stack, and sends them to the renderer. The renderer parses the HTML documents, applies the CSS, executes the JavaScript, and sends the parsed frame back to the browser to be painted and displayed. The sandbox and process isolation rely on the operating system's security mechanisms. For example, the Chrome sandbox on Linux relies on user namespaces to restrict the sandboxed process's access to resources and Seccomp BPF to restrict system calls. The operating system prevents memory access into other processes and the sandboxed process from accessing the host's file system and other capabilities.

Site Isolation is a browser security architecture that enforces isolation between web applications on the process level by placing only the content belonging to the same site in the same sandboxed renderer process [2]. A *site* is defined as a scheme and a registrable top-level domain, also called an extended top-level domain plus one domain part (eTLD+1). For example, the site `https://example.com` would comprise all subdomains like `https://*.example.com`. The granularity of a site is coarser than that of an origin because it ignores both subdomains and ports. Some URLs, for example `data:` URLs, are treated as *opaque origins*. They possess a unique site and origin that never match another site or origin. For each site, the browser process spawns a new renderer process to process only the documents of this site, as shown in Figure 1. In the following, we will cover the three vulnerabilities that Site Isolation mitigates.

Spectre Spectre exploits speculative execution and architectural side channels in modern CPUs to potentially leak a process's full memory space [1]. Agarwal et al. [6] showed that Spectre attacks could be mounted just by executing JavaScript in the browser to leak the renderer's memory. Since Spectre cannot leak data from other processes, Site Isolation mitigates

the vulnerability by ensuring that all potentially leaked data belongs to the attacker's site.

Renderer Compromise An attacker who has exploited a memory bug to achieve remote code execution in the renderer can leak the renderer's whole process space, including all web application data, such as cookies. Site Isolation also mitigates this vulnerability by placing only the attacker's site data in the process that the attacker can compromise.

Universal Cross-Site Scripting Furthermore, Site Isolation mitigates Universal Cross-Site Scripting vulnerabilities that allow the execution of malicious JavaScript in the context of other web applications in the victim's browser. Site Isolation achieves this by isolating sites on the process level and providing a cleaner architecture with explicit domain bounds, well-defined IPC interfaces, and centralized security checks that prevent coding errors leading to UXSS bugs. Reis et al. [2] determined that Site Isolation mitigated all previous UXSS vulnerabilities in the Chrome browser.

B. Site Isolation Implementation

In this section, we cover the specific programming paradigms that are required by Site Isolation and examine their impact on the browser architecture.

Inter-Process Communication All renderer processes communicate with the browser process. In addition to providing access to the network stack and file system, the browser process also passes messages between the renderers to support cross-site communication. Both Chrome and Firefox use Chrome's Mojo library [7] for inter-process communication. The implementation of the underlying IPC connection varies depending on the operating system, but generally relies on shared memory. Mojo multiplexes many channels over one concrete IPC connection. This minimizes the overhead of creating additional channels and encourages an architecture that splits the communication between the browser and renderer processes into many topic-related interfaces. However, the browser process must keep track of the corresponding site for many IPC channels with different renderers.

The resulting architecture is more complicated than Figure 1 conveys. Figure 2, which shows Chrome's Site Isolation architecture, depicts the different IPC channels required solely to control three frames in two different renderers. The RenderProcessHost and RenderFrameHost components on the left control the RenderProcess and RenderFrame components on the right, with messages passed via the IPC channel. Both the browser and renderer processes keep track of the frame tree. The frames in the renderer handle the HTML documents.

Service Processes If the host system has enough memory, modern browsers also move parts of the browser process into less privileged processes. Chrome, for example, creates separate processes for the Network Service, Storage Service, and GPU processes. By defining a sandboxed policy tailored to each process, the Chrome developers further mitigate the impact of memory bugs in one of the services. These privileged processes communicate directly with the renderers, thus also increasing the complexity of IPC.

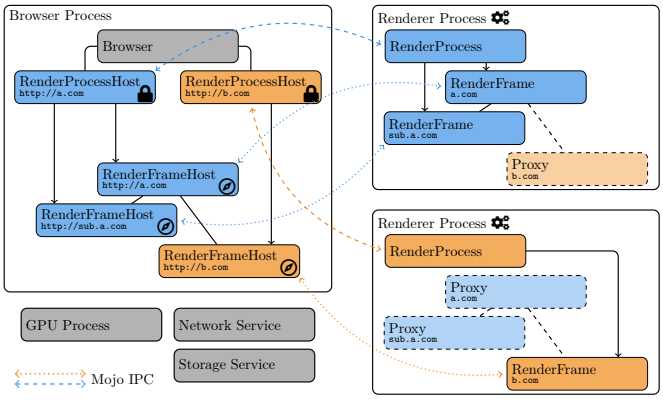

Fig. 2. Chrome's site isolated multi-process architecture

As a compromised renderer can send arbitrary IPC messages, the browser process must implement checks to handle this. To enforce the same-origin policy and prevent a compromised renderer from accessing confidential data from another site, the browser process must check the renderer's site during every interaction. The browser process ensures that a renderer can read any resource it passes to the renderer and be permitted to send any request or message it sends on the renderer's behalf. To achieve this, the browser process must keep track of each renderer's site and all permissions that the user might have granted to that particular site.

Process lock As soon as the renderer processes the first content, the renderer must be assumed to be potentially malicious. The first input to the HTML parser or JavaScript engine might exploit a memory bug in the parser or the JS engine's JIT compiler and compromise the process. Thus, the browser process assigns the site (🔒) to the renderer, "locking" the renderer process to this site. This site cannot change during the process's lifetime. For example, consider the RenderProcessHost in blue in Figure 2, which is locked to `a.com` in the browser process. As this information is stored in the browser process, it can be used for security checks without the risk of tampering.

Oftentimes, there exist several values that can be used for security checks. Chrome, for example, also saves the latest document origin (🧭) for every frame in the browser process.

Site-bound channels To reduce the overhead of passing all IPC messages through the browser process, the browser process can establish a direct IPC channel between two parties, for example, a renderer and the network service. The browser process lets the privileged party, in this case, the network service, create the channel and passes one end of the channel to the renderer. In doing so, the browser process communicates the renderer's site to the network service. The network service saves the site and its end of the channel and conducts all security checks based on this value. This allows the network service to communicate directly with the renderer while profiting from the process lock.

Killing compromised renderers The browser process kills renderers if it receives malformed or invalid messages indicat-

ing a renderer compromise. It contains the compromise to the renderer and reduces the susceptibility to undefined states in the browser process that could lead to Site Isolation bypasses.

Related Security Mechanisms Site Isolation is complemented by Cross-Origin Read Blocking, Cross-Origin Opener Policy, and Cross-Origin Embedder Policy. Cross-Origin Read Blocking (CORB) is a mechanism in the browser process that prevents the leaking of SOP-exempt cross-origin resources to the renderer. For historical reasons, resources requested by `<img>` and `<script>` tags were exempt from the SOP. Since resources of certain file types are invalid in these contexts, CORB blocks these resources. The Cross-Origin Opener Policy (COOP) allows web servers to request origin-based isolation for their documents. The Cross-Origin Embedder Policy (COEP) allows servers to define which sites may embed their resources. While our proposed sanitizer can detect the leaks that follow CORB bugs, our research does not focus on CORB, COOP, or COEP bugs; our fuzzer does not aim to trigger them.

Site Isolation Deployment Status Site Isolation is active since 2018 in Chrome [2] and 2021 in Firefox [3] respectively. However, it is inactive on devices with less than 2GB of RAM and Android WebViews. Safari does not isolate iframes of different sites in different renderers at the time of writing. The WebKit developers are working on integrating Site Isolation. However, the implementation poses a significant engineering effort [2].

C. Site Isolation Bypass

Site Isolation bypass vulnerabilities allow an attacker to circumvent the Site Isolation and access the data of another site. Site Isolation bypass bugs facilitate attacks on all other web applications running in the victim's browser by executing malicious JavaScript in the context of the other website, i.e., achieving UXSS or stealing confidential data such as cookies. Therefore, they are more dangerous than Cross-Site Scripting attacks, which only facilitate attacks on a single vulnerable website. While Site Isolation bypass vulnerabilities cannot be exploited to compromise the victim's host system or access local files, potentially mounting attacks on all web applications that the victim uses is a powerful capability.

Site Isolation bypasses are caused by semantic bugs in the functions that the browser process uses to track and determine the site of a renderer or in the security checks based on this tracked value. In contrast to memory bugs (e.g., buffer overflows), the Address Sanitizer cannot detect semantic bugs. Instead, they require an oracle that predicts the correct program state to compare to the observed state. Furthermore, these semantic bugs are hidden deep in the application logic, and triggering them requires a semantically valid initial state and a sequence of syntactically and semantically valid IPC messages. Invalid messages detected by the browser process lead to immediate renderer kills.

Attacker Model We assume that the attacker has already achieved RCE in the sandboxed renderer. The attacker can execute arbitrary code in the renderer process, read the whole process memory, and send arbitrary IPC messages to other processes. Since the attacker can execute arbitrary code in the renderer, they can also circumvent all renderer-side security checks.

We argue that this attacker model is a realistic assumption since the DOM engine and JavaScript engine remain prone to memory bugs [8]. Especially, the number of discovered JIT bugs in the JS engine remains high [9]–[11].

TABLE I
SI BYPASS VULNERABILITIES CLASSIFIED BY CAUSE

Class	Description	#Bugs	Example
1	Missing Checks	28	CVE-2018-18345
2	Bypassed Checks	4	CVE-2020-6385
3	Origin Confusion	6	CVE-2022-1637

III. Vulnerability Analysis and Classification

We analyze all bug reports of previous SI bypass vulnerabilities in the Chrome and Firefox browsers to identify common vulnerability causes and create a classification of SI bypass vulnerabilities. We manually examine every bug from the Chromium bug tracker with the tag *Internals>Sandbox>SiteIsolation*. For Firefox, we analyze the bugs in the Site Isolation meta bug trackers [12]–[14]. In addition, we examine every CVE entry in the NVD for one of the browsers whose description contains the term *Site Isolation*. For both browsers, we only consider bugs filed after Site Isolation was rolled out to filter progress trackers created during the implementation of SI. From 1,328 examined bug reports, 39 described vulnerabilities that facilitated SI bypasses. Table V in the appendix lists all of these previous SI bypass vulnerabilities in Chrome and Firefox.

SI Bypass Classes The basic workflow for secure interactions between the browser and the renderer process requires the browser process to apply security checks to every request of the renderer. To conduct these checks, the browser process compares the origin or URL that the renderer claims to represent or of any resource that the renderer requests against a secure value (e.g., from the process lock). There are three points of failure in this workflow: the security check might be missing, the renderer might circumvent the check, or the browser might confuse the secure value. Table I overviews the three vulnerability classes.

We revisit all known SI bypass vulnerabilities and assign them to one of the three classes. Furthermore, we aim to identify the renderer behavior required to exploit them. The browser developers accept proof-of-concept exploits for SI bypass vulnerabilities that include manual patches of the renderer code to simulate the behavior of the compromised renderer. Thus, we can quickly identify the behavior that a fuzzer must simulate to trigger SI bypasses.

A. Missing Checks

This is the most common class of SI bypass vulnerabilities. If the browser process lacks security checks to verify that a

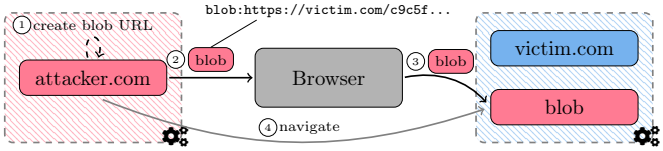

Fig. 3. Schematic view of the CVE-2018-18345 SI bypass

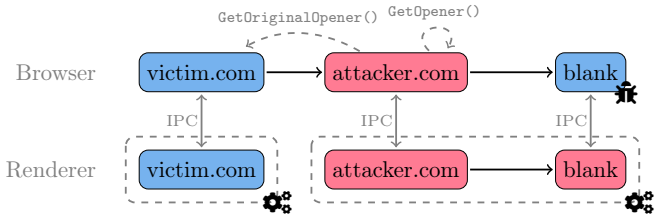

Fig. 4. Schematic view of the CVE-2022-1637 bug

renderer belongs to the claimed site, a compromised renderer can spoof parameters in IPC messages. Vulnerabilities also emerge if security checks are only conducted on the renderer process's side. Once the attacker compromises the renderer, they can freely change the control flow to circumvent security checks. Thus, only checks in the browser process can prevent this vulnerability class. To exploit these bugs, the compromised renderer must spoof origin-related parameters in IPC messages. Other parameters were rarely used for exploits. In addition, the compromised renderer must bypass all renderer-side security checks. The remaining section provides an exemplary case study of CVE-2018-18345 from this class.

Case Study: CVE-2018-18345 This vulnerability in Chrome before version 71.0.3567.0 allowed compromised renderers to register an HTML document with malicious JavaScript as a blob URL of another site. Upon navigation to the blob URL, the malicious script was executed in the context of the victim site. The IPC blob registry interface that the browser process provides to the renderer accepts a blob URL in the common UUID form. However, the browser process did not verify that the host of the provided blob URL matched the site of the renderer process. Figure 3 details the whole attack flow. The attacker creates the blob URL ① and registers it for the victim site by spoofing the host in the URL ②. The browser process saves the blob URL in the context of the victim site ③. It is executed when the attacker navigates to it ④.

B. Bypassed Checks

We found four bug reports for vulnerabilities that emerged because security checks existed but were faulty and could thus be circumvented by a compromised renderer. The causes for this vulnerability class are diverse. A frequent cause for such vulnerabilities is the renderer outliving the corresponding control structures in the browser process and the browser subsequently skipping security checks.

Case Study: CVE-2020-6385 Security checks were added to the blob URL store in response to the previously discussed SI bypass vulnerability with *CVE-2018-18345*. However, the checks were not applied if the renderer process was shutting down because the control structures in the browser process holding the process lock might have been deleted.

A compromised renderer could spoof a frame detachment IPC message to the browser process. This would trick the browser process into believing that the last frame from the renderer was removed and the renderer could be evicted. If the compromised renderer ignored the SIGTERM signal sent by the browser process, it would outlive the control structures in the browser process. Thus, the compromised renderer could send the same spoofed blob URL IPC message as in *CVE-2018-18345* because the provided URL would not be checked.

C. Origin confusion

If the site value that the browser process uses for security checks is wrong, the browser process will accept spoofed origin values from compromised renderers. This origin confusion is oftentimes triggered by complex cross-site navigation. To exploit these vulnerabilities, a compromised renderer must first trigger the origin confusion by combining various browser navigation API routines and then spoofing origin parameters in IPC messages. It follows an exemplary case study of CVE-2022-1637 from this class.

Case Study: CVE-2022-1637 This vulnerability in Chrome up to version 100 allowed a compromised renderer to spoof the origin of the top-level frame and, for example, access the cookies of a cross-site document framing the attacker's site. A new iframe or window created with a blank URL inherits the origin of the opener to facilitate communication between the two. The browser process contained a bug because the wrong method was used to retrieve the origin of the newly created frame. Figure 4 shows the frame trees in the renderers and the browser process with the IPC channels between the frame objects. The used method, `GetOriginalOpener`, returns the origin of the top-level frame instead of the current frame. On the renderer side, the frame's origin was derived correctly, and the frame object was placed in the renderer process of the opener. Thus, if `victim.com` frames `attacker.com`, `attacker.com` could open a new window to a data URL with `_blank` target to trigger the origin confusion in the browser process. The browser-side frame object would be associated with the victim site. The renderer-side object would be correctly associated with the attacker site. A compromised renderer could then request the victim site cookies via the IPC channel of the blank frame and spoof the origin parameter of the request to match `victim.com` to steal the cookies of the victim site.

IV. Fuzzer Design

Our analysis of the SI bypass proofs-of-concept revealed them predominantly relying on three elements: complex cross-site navigations to trigger origin confusion, circumventing renderer-side security checks, and spoofing origin-related IPC parameters. We propose a fuzzer architecture fulfilling all three requirements to trigger SI bypass vulnerabilities.

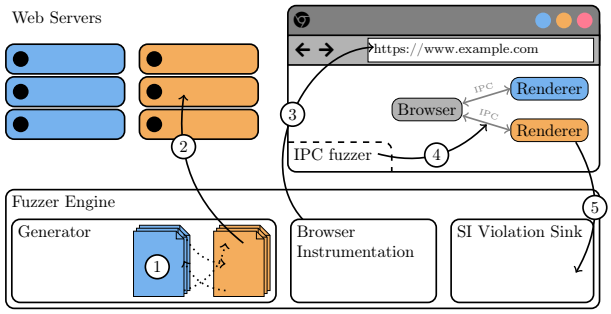

Fig. 5. SI bypass fuzzer

We can trigger cross-site navigations by generating HTML documents that invoke the browser's navigation APIs. To circumvent renderer-side checks and spoof the contents of IPC messages, we patch the code of the renderer process. We add methods to turn off renderer-side checks per renderer process and to modify the origin parameters of the following IPC message sent by the renderer. This approach is similar to the fault-injection technique employed by Bars et al. [15]. We thereby simulate the behavior of a compromised renderer.

In contrast to memory bugs that the Address Sanitizer can detect, SI bypass bugs are semantic bugs, and we need an oracle to detect them. The oracle must detect the execution of attacker-provided scripts in the victim context and leaks of critical data from the victim context to the attacker context. We execute a full browser to process HTML documents from different sites. In our setting, one of the sites is the attacker site, which has access to the capabilities of the compromised renderer, and the other site is the victim site whose data the attacker wants to access. Since we know the correct site of each document and which site the sensitive data belongs to, we can detect Site Isolation violations by checking whether we execute on a site that should be isolated or can access data belonging to a different site.

Overview Figure 5 provides an overview of our fuzzer. First, the generator creates a set for cross-referencing HTML documents with JavaScript content for each site. These documents are then pushed to two servers with different IPs and, therefore, different sites. In the third step, we navigate the instrumented target browser to the sites and observe the behavior. As the renderer processes the document and communicates with the browser process, the IPC fuzzer module intercepts the messages sent by the compromised renderer and randomly modifies origin-related parameters. Last, our sanitizers detect successful SI bypasses and report them to the fuzzer.

A. IPC Message Manipulation

We implement an IPC fuzzer component in the native code of the renderer to intercept and manipulate outgoing IPC messages. The IPC fuzzer targets site-related parameters in outgoing messages and randomly mutates them to simulate the behavior of a compromised renderer.

Both Chrome and Firefox define their different IPC interfaces via interface definition files. They employ a service-oriented architecture: each interface contains a group of related functions that either the renderer or browser process offers to the other. IPC calls are similar to calling a function. They accept several arguments and may have a return value.

Relevant Data Types We manually examined the IPC interface definitions of the two browsers to identify the types of parameters that a compromised renderer might spoof to bypass Site Isolation. Chrome defines structured types to transmit origin-related parameters. There are three atomic parameter types: The *url* type represents a standard URL. It comprises a scheme, user info, host, port, path, query, and fragment. The *origin* type consists of scheme, host, and port. The *schemeful site* type contains a scheme and an eTLD+1. In addition, there is one composite type. The *storage key* contains an origin, a schemeful site, and an ancestor chain bit. We examined all IPC interfaces for navigation and storage activities to ensure that these are the only types used to transmit site-related information between processes. Therefore, we can reliably detect all site-related parameters by type.

Our examination of the Firefox IPC interfaces revealed that Firefox processes only exchange three different types of site-related data: URL, origin, and domain. In addition, no structured IPC parameter types exist for these parameters in Firefox. Instead, the processes exchange URLs and origins in string format. Thus, we cannot use types to determine if a parameter contains site information. Instead, we use the parameter identifier to determine if a string parameter is site-related, searching for identifiers containing the words URL, origin, domain, or spec. Like Chrome, Firefox often exchanges URL and origin strings as part of more considerable struct parameters.

Mutation Operations We identified two meaningful mutation strategies for site-related IPC parameters based on the proofs-of-concept of known vulnerabilities. The easiest mutation is to take another URL and replace all components of the parameter with components of the URL. The IPC message triggered by the 'history.replace()', for example, contains the URL of the new history entry. The replacement mutation overwrites this URL with another random URL, possibly with a different host or scheme. This mutation can also be applied when the site or origin is opaque (i.e., does not have a scheme and host).

The second mutation type is to replace just the host value of the parameter. This mutation allows for easy exploitation of SI bypass vulnerabilities from missing checks because the remainder of the URL contains relevant data. An attacker would replace his own host in the IPC message sent by the renderer process with the host of the victim site. The exploit for CVE-2018-18345 displayed in Figure 3 is an example of this technique. The attacker sends an IPC message to create a blob URL but replaces his host in the URL parameter with the victim host to trick the browser process into moving the malicious blob to the victim process.

Input Synchronization Our fuzzer produces two different kinds of inputs: the HTML documents that make the base behavior of the browser, and the mutations of the intercepted IPC messages that simulate the compromised renderer. To

reproduce the bugs that our fuzzer discovers, the fuzzer must reliably replay the exact IPC mutations at the right time during document processing. Thus, we need to synchronize the execution of the cross-site interactions triggered by the document and the scheduling of IPC message mutations. We achieve this by combining both inputs in one document: We encode IPC message mutations in the HTML document as invocations of a custom browser API.

The IPC interceptor collects message mutation instructions in a FIFO queue. The custom browser API has functions for each IPC parameter type (URL, origin, schemeful site, storage key) and each mutation type (replace fully, replace host) that take a replacement value and enqueue an instruction to apply the respective mutation. Whenever the interceptor intercepts an origin-related value, it checks the head of the queue for a matching mutation instruction. The interceptor dequeues and applies a matching instruction. Otherwise, it transmits the IPC parameter unmodified.

The combination of enqueued message mutation instructions and regular JavaScript statements robustly encodes the behavior of the IPC interceptor and produces reliable proof-of-concept exploits on fuzzer crashes. We evaluate this input format by re-implementing four proofs-of-concept from known SI bypass vulnerabilities using our IPC fuzzer API instead of manual renderer patches. For all four vulnerabilities, we could implement minimal and robust proofs-of-concept.

Renderer Kills The browser process applies sanity checks to the incoming IPC messages and their parameters. If the browser process detects a malformed message or invalid parameters, it infers that the renderer's behavior deviates from its implementation, indicating a compromise, and kills the renderer process. This regularly produces an error in the browser instrumentation, requiring a time-intensive restart of the whole browser. Thus, the renderer kills severely impact the efficiency of our fuzzer. We experimented with mutating other parameters of IPC messages, but this led to a stark increase in renderer kills without visible benefits for SI bypass detection. We discuss techniques to reduce the impact of renderer kills in Section VII.

B. Input Generator

The generator aims to generate HTML documents that lead to diverse cross-site interactions. The generated documents should be syntactically and semantically valid and cover the whole browser API. Syntactical and semantic errors lead to JavaScript exceptions that stop the execution of the statement, rendering it useless. Thus, we require type information for the browser API to produce valid invocations of its functions.

Manually written grammars for browser API fuzzing often do not cover the whole API. Nevertheless, if the document coverage of the browser API is incomplete, we might miss vulnerabilities related to the missed APIs. Thus, we require a complete grammar.

Web IDL The W3C standardizes the web browser's JavaScript interface in a definition language called Web IDL. Every property of the browser API, its inheritance, member functions, attributes, and visibility are defined in the definition files. In contrast to the information available to JavaScript running in the browser, the Web IDL files contain types for all method signatures and attribute definitions. We can ensure type validity for our generated documents by leveraging the information from the Web IDL files.

Algorithm 1 Random JS generation algorithm

Require: G : JS grammar
1: $s \leftarrow \emptyset$ ▷ *state of variables*
2: $n \leftarrow 0$
3: **while** $n < 20$ **do**
4: **if** RAND $< \frac{\lfloor s \rfloor \cdot 10}{|G|}$ **then** ▷ *rand* $\in [0, 1]$
5: $obj \leftarrow$ WRANDCHOICE(s)
6: **else**
7: $class \leftarrow$ WRANDCHOICE(G)
8: $obj \leftarrow$ INSTANTIATE$(class, G, s)$
9: ADD(s, obj)
10: $members \leftarrow$ MEMBERS(obj)
11: $m \leftarrow$ WRANDCHOICE$(members)$
12: $p \leftarrow$ GENPARAMS(m, G, s)
13: $newobj \leftarrow$ INVOKE(obj, m, p)
14: ADD$(s, newobj)$
15: $n \leftarrow n + 1$

Both Chrome and Firefox use the Web IDL files during the pre-build step to automatically generate the JavaScript bindings of the renderer. Both browsers have slightly different Web IDL specifications because they are not entirely compliant with the other HTML specifications. Parsing the corresponding Web IDL files of the fuzzed browser, we create a grammar that perfectly fits the specific browser to the particular version. We supply a small handwritten grammar to supply the signatures of the JavaScript built-in classes.

While the Web IDL specifications contain the possible values for function string parameters that expect specific keywords in the form of enum definitions, they do not include this information for the keywords passed to the attributes of HTML elements. These keywords are only loosely defined in the HTML standard or the browser's source code. However, the MDN web docs list the keywords for every HTML attribute in a structured format that we can parse to extract this information. We also supply a manually written map from HTML tag names to the respective DOM API class names.

JavaScript Generation Similar to other fuzzers creating JavaScript code [10], [16], we generate code in static single-assignment form (SSA). For each variable, we only assign a value once, during its declaration. Thus, each variable is valid in all statements after its declaration and keeps its initially assigned type. Using the SSA form, we implement a context-aware generator that tracks the current context of available variables and their types to reuse them in complex statements.

We provide the pseudo-code of our generator in Algorithm 1. In each iteration, the generator chooses a random object from the current context or a random class from the grammar and instantiates an object of this class. Next, the generator chooses a random object member and generates the JS code to call the member function or assign a value to the attribute. The generator either uses fitting variables

from the context for the required parameters or instantiates the required values or objects, preferably by using members of existing objects. For objects that cannot be created by calling a constructor, the generator checks if the object can be obtained as a property of the browser API or if a fitting object is returned by any function that can be invoked. We allow the creation of helper objects to obtain the required objects from a member function call up to a recursion level of two. The parameter generation method also creates functions for callback or event handler parameters.

Boost Object Reuse & Navigations We raise the probability of reusing existing objects from the scope to increase coherence between the generated statements and create complex API interactions. We also increase the number of cross-site navigations because they are the main precondition for triggering origin confusion SI bypasses. To this end, we employ a weighted random algorithm with a probability for the preference set of 0.2, similar to Kim et al. [16]. We manually selected all navigation-related interfaces from the browser API to choose them with a higher probability during the initial object selection step. Since most of these interfaces define and inherit many members unrelated to navigation, we also select a preference set of navigation-related members.

The generator produces random primitive values of matching type whenever it encounters them in parameters. Optional, nullable, or variadic parameters are populated or left empty at random. For URL strings, it inserts either the URL of one of the generated fuzzing input websites or a unique URL belonging to one of the following categories: Data, blob, or `javascript:` URLs. We limit the nesting documents to a maximum depth of two to prevent infinite recursion.

Service Worker Service Workers have access to a set of powerful capabilities. They can, for example, intercept all outgoing HTTP requests. To fuzz this additional interface, the generator also creates JavaScript files with the populated callback functions of a service worker. Our Web IDL-driven approach allows us to quickly generate code that utilizes the API available to the Service Worker by filtering by the visibility scope attribute of the browser properties. The Service Workers are registered by a short code snippet that the generator adds to each test case.

User Interaction Some exploits require user interaction (e.g., clicking a link) to succeed. We instrument the browser under test with Playwright to automatically interact with the websites and click on buttons and links. We utilize Playwright because it prove to be more stable in the context of high process counts and renderer kills.

C. Site Isolation Bypass Detection Oracles

We now describe the detection mechanisms of our new SI bypass bug oracles. Since the browser executes renderer processes for both websites, we can observe the data flows triggered by the HTML documents and the IPC message mutations. We want to detect three different forms of Site Isolation bypasses: execution of attacker-provided JS in the victim renderer (e.g., UXSS), leaks of critical victim site data to an attacker-controlled renderer process, and cross-site process-reuse. Cross-site process-reuse does not produce cross-process data transmissions. But data residing in the process is vulnerable to Spectre attacks.

Process sanitizer This sanitizer detects both UXSS and cross-site process-reuse. We implement the process sanitizer as a function that we add to the browser API. Our generator knows the correct site of the document it is creating. It inserts invocations of the process sanitizer function into all documents and passes the correct site as a parameter. On its first invocation, this process sanitizer function tags the site received as a parameter to the process. On every following execution, the process sanitizer compares the passed site to the tagged site. Thus, it detects whenever a renderer is reused between different sites, either because it is erroneously shared or because of UXSS.

Leak flow sanitizer The leak sanitizer detects data from the victim web application that leaks into the attacker's renderer process. The leak sanitizer is activated for the whole renderer process by calling a JS function from any script. From this point, the sanitizer examines all incoming IPC messages for a known magic value. When generating the documents for the victim website, the generator randomly produces this value when generating strings. Thus, the victim page passes this magic string to many different browser APIs. In addition, we visit a seed page hosted by the victim web server after every browser restart. The seed page contains scripts that store the magic value in cookies, local storage, file systems, and the IndexedDB of the victim web application. Whenever the fuzzer triggers a SI bypass bug that leaks data of the victim process, the leak sanitizer detects the leak and raises a warning.

While the generator would randomly produce the proper browser API invocations to exfiltrate data, we increase the chance of detecting a successful exploit by calling a predefined sanitizer JavaScript function at the start of every function in the generated documents. This sanitizer function aims to exploit a triggered SI bypass vulnerability and exfiltrate data from the other site, thus producing a data flow detected by the leak sanitizer. The function reads data from cookies, local storage, file system, and the indexed DB of the web application. Figure 9 (appendix) contains an example for the output of the generator.

To detect CORS vulnerabilities, we detect if the reply to a credentialed cross-site request is passed to the compromised renderer process. We achieve this by providing two additional HTTP endpoints that are fetched by the sanitizer function. The first endpoint mimics a CORS-protected resource with cookie-based authentication. It reflects all received cookies in the HTML document of the reply, setting the Access-Control-Allow-Origin header to only allow the victim's origin. Thus, if the compromised renderer can access the resource even though it is not permitted, the leak sanitizer detects the magic value of the cookie reflected in the reply. The second endpoint simulates a non-credentialed CORB-protected

response. It does not require cookies to be set and returns an HTML document containing the secret value. We set the Content-Type to `text/html` to enable CORB. If the browser does not block the reply, the leak sanitizer detects the magic value in the document.

V. Fuzzer Implementation

We implement the generator and browser instrumentation in Python and the IPC fuzzer as a small submodule in the C++ codebase of Chrome, Firefox, and WebKit (Safari). This section details the specific implementation choices for each of the components.

While we successfully implemented the IPC fuzzer for WebKit, we could not instrument Safari combined with the patched WebKit because Safari's Webdriver does not support it. We discuss this in Section VII. Consequently, this section will focus on the implementation for Chrome and Firefox.

A. Browser Instrumentation

We use Playwright to instrument the two browsers and automatically interact with the generated web pages. Playwright controls the Chrome browser via the Chrome DevTools Protocol endpoint. Thus, we can exchange the Chrome browser bundled with Playwright for our patched version. To instrument Firefox with Playwright, some patches are required that we manually apply to the code base, together with our IPC fuzzer patches.

The browser vendors see compromised renderer processes as a threat. Consequently, safeguards are in place that terminate the renderer process if the browser process detects a malicious IPC message. Thus, the browser regularly kills the renderer process in our experiments, decreasing the stability of the browser instrumentation. We must pay special attention to handling these errors and cleaning up the remaining processes to prevent the fuzzer from freezing up.

Configuration The default configuration of Playwright disables Site Isolation in both browsers, leading to meaningless SI bypasses. We activate Site Isolation by overriding the command line flags that Playwright passes to the browser or by patching the configuration files. Furthermore, we activate browser features like Chrome's out-of-process network service that is relevant in the context of Site Isolation but disabled by Playwright by default.

Visit Strategy The browser executor starts the browser and initially visits the seed pages of both sites to populate cookies and storage. It then visits the victim and attacker pages to simulate the browsing behavior of a person lured to the attacker's page, e.g., via phishing. On both pages, the executor interacts with every iframe and clicks every button and hyperlink to trigger manual cross-site navigations.

Build Configuration We additionally compile the browsers with Address Sanitizer instrumentation to detect memory bugs in the browser process. In particular, we are interested in crashes of the browser process that indicate sandbox escape vulnerabilities, discovered as a byproduct of our fuzzing campaign. Furthermore, we discovered that the debug assertions that are generally used in browser fuzzing builds to detect bugs [17], [18] are detrimental to our fuzzer: The debug assertions in the renderer and the browser process detect many symptoms of a compromised renderer process early and terminate the renderer. Therefore, a browser with debug assertions is artificially robust against SI bypasses and produces significantly more renderer kills, reducing our fuzzer throughput. Since debug assertions can be circumvented like any renderer-side check, they provide no security in real-world scenarios, even if someone uses a debug build for regular browsing. Consequently, we turn off debug assertions.

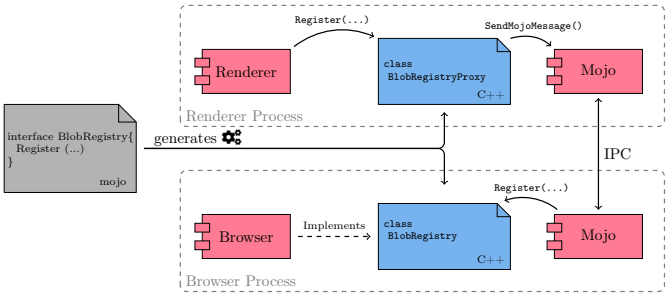

Fig. 6. Mojo IPC bindings

B. Fuzzer Hooks

The IPC fuzzer we implement in Chrome and Firefox consists of two components: the main class that stores instructions from the JS API in a queue and the fuzzer hooks that intercept every site-related parameter in outgoing messages and apply mutations from the queue. Both browsers rely on the Mojo IPC library, developed as part of the Chromium project, to multiplex many logical IPC channels over one real IPC connection via shared memory. Because of the low cost of additional IPC channels, both Chrome and Firefox define many IPC interfaces through which the browser and renderer processes communicate.

All IPC channels merge at the point where the serialized messages are written to the same concrete connection, over which the channels are multiplexed. However, we cannot intercept and modify the messages at this point because the message is a byte array that must be deserialized before type information is available. Another option would be to manually patch every function that sends IPC messages to the browser process. Due to the number of functions involved, this involves significant effort. Additionally, the frequency of changes to the browser's code base makes transferring our changes between browser versions infeasible.

Fuzzer Hook Generation Instead of manual patches, we propose an automatic solution to insert the IPC fuzzer hooks that exploits the browser's build process. Both browsers define IPC interfaces in the form of interface definition files. These files are parsed during the pre-compilation step of the browser build process, and C++ bindings are generated for both endpoints that provide a layer of abstraction around the Mojo IPC library.

Figure 6 provides an example of this setup: The `BlobRegistry` IPC interface of the Chrome browser. The interface is defined in a Mojo IDL file. The Mojo parser processes this file and creates C++ files with two classes: a proxy class for the client that uses the interface and an abstract class for the service that provides the interface. The proxy class contains a method for every defined IPC message that wraps the provided parameters in a message, serializes it, and sends it via the linked Mojo library. On the other side, the process that provides the interface must implement the abstract class, overriding all virtual handler methods. In this process, the Mojo library receives and deserializes the message and calls the matching handling method. Return values of the handler method are sent back via IPC and returned to the client as a return value of the invoked method of the proxy class.

We modify Chrome's bindings generator to automatically inject a fuzzer hook into the generated classes for every site-related parameter. These fuzzer hooks act as callbacks into the IPC fuzzer class, which tries to apply an enqueued mutation and overwrites the parameters. Firefox only uses the Mojo library for the low-level IPC and employs a different IDL and IDL parser. Therefore, we patch the Firefox IPC binding generator in the same fashion.

We achieve coverage over all IPC messages on all interfaces by patching the binding generators. All IPC parameters of the relevant types are reliably intercepted. In addition, we can update to newer browser versions simply by reapplying our patches. Changes to both binding generators are infrequent; thus, our patches apply without conflicts.

C. Additional Browser Patches

In addition to renderer patches in the automatically generated code of the IPC bindings, we manually apply patches to implement our sanitizers.

Bypass Renderer-side Checks Similar to the security checks in the browser process, security checks in the renderer assert that the renderer can conduct an operation on behalf of a specific origin and terminate the renderer in case of an assertion failure. Once the attacker has compromised the renderer, they can arbitrarily change the control flow of the process. Thus, they can also circumvent all security checks implemented in the renderer. If left unmodified, the renderer-side checks prevent our fuzzer from triggering SI bypass bugs because the renderer terminates itself. Thus, we manually patch the renderer to add a *compromised* mode that overrides all renderer-side security checks. We expose a browser API invoked by our attacker site to set a boolean flag to activate the mode, assuming the renderer can be compromised as soon as it evaluates attacker-provided JavaScript code.

Leak Sanitizer We also implement the Leak Sanitizer as a renderer patch. We patch the deserialization function that processes incoming IPC messages. The sanitizer is activated by calling a function exposed to the JavaScript API. From this point on, it examines every incoming IPC message. It searches for the known byte sequence of the magic string in the binary blob of the serialized message. Thus, the sanitizer does not depend on knowledge of the message format. Since both Chrome and Firefox use Mojo for the lowest level of IPC, we can apply the same patch to both browsers.

VI. Evaluation

In this section, we present the results of our evaluation and the month-long fuzzing campaign targeting Chrome and Firefox. We evaluate our fuzzer in three steps: First, we examine the semantic validity of the input documents created by the generator. Second, we evaluate the capability of our fuzzer to trigger and detect SI bypass bugs on old versions of Chrome with three known vulnerabilities. Finally, we measure the code coverage achieved by our fuzzer and compare it to that achieved by the UXSS fuzzer Fuzzorigin [16]. We execute all three evaluation steps on a system with Debian 12, an AMD Epyc 7713 processor with 64 cores and 64 GB of memory.

A. Semantic Validity

The semantic validity of the generated JS code is important because semantic errors terminate script execution early, leading to ineffective fuzzer iterations. While the generator can ensure syntactic validity during the lowering step to JS code, semantic validity is more challenging. We can prevent semantic errors from aborting the whole script by guarding each statement with a try-catch block. In contrast to JIT-fuzzing, where try-catch blocks break JIT optimizations and thus cannot be used [10], try-catch blocks have no adverse side effects for our fuzzer. However, all the following statements that depend on a variable defined by the erroneous statement will also fail.

We also leverage these try-catch-finally blocks to measure the semantic validity of the generated statements. Each catch- and finally- block emits a console log that indicates that a block of the respective type was executed. The fuzzer collects the console logs and counts the number of executed catch- and finally- blocks. This enables us to compute the number of executed statements that led to an exception.

Our fuzzer creates different inputs for Chrome and Firefox because it uses the respective Web IDL definitions of the browsers. Thus, we evaluated our fuzzer on both Chrome and Firefox. We run our fuzzer for 24 hours each on both browsers and measure the fraction of successfully executed statements. In Chrome, 89.5% of executed statements pass without exceptions. The number of exceptions in Firefox is slightly higher, 85.3% of statements are semantically valid. The noticeable difference in validity can be explained by the generator utilizing the Web IDL definitions of the respective browser to generate the inputs. The browser developers use different extended attributes to express additional semantics of the JavaScript APIs. Our fuzzer might lack support for some extended attributes, thus mistakenly generating invocations of unavailable interfaces.

B. Evaluation on known bugs

To evaluate our fuzzer's capability to trigger and detect SI bypass bugs, we run the fuzzer on old browser versions

TABLE II
KNOWN VULNERABILITIES IN CHROME USED FOR FUZZER EVALUATION

Vulnerability	Chrome Version		Class
	Vulnerable	Evaluated	
CVE-2022-1637	< 101.0.4951.64	99.0.4844.84	3
CVE-2019-5856	< 76.0.3809.87	67.0.3396.99	1
CVE-2018-18345	< 71.0.3578.80	67.0.3396.99	1

TABLE III
ORACLE EVALUATION ON KNOWN POC'S

ID	Class	LeakSan	ProcessSan
CVE-2018-16074	3	○	●
CVE-2019-5773	1	●	○
#40093844	2	●	○
CVE-2024-1671	3	●	○
CVE-2022-3044	1	○	○

with known vulnerabilities and measure the time required to trigger the bugs. Since our fuzzer requires browser patches, we must adapt these patches and apply them to the old browser versions. To minimize the engineering overhead, we try to identify browser versions that can be used to reproduce several known bugs simultaneously. Table II lists the bugs used for evaluation and the vulnerable and evaluated browser versions.

We covered the bugs with CVE-2022-1637 and CVE-2018-18345 in Section II-C as examples for their respective classes. CVE-2019-5856 is another vulnerability of class 1 caused by missing security checks in the browser process. In this case, a malicious renderer could access `filesystem:` URLs of any site because origin checks only existed in the renderer process. We can reproduce the three bugs with two old Chrome versions. The old browser versions are incompatible with recent Linux versions; thus, we run them in Docker containers with old versions of Ubuntu, i.e., Ubuntu 18.04 for Chrome version 99 and Ubuntu 14.04 for Chrome 69, respectively.

Surprisingly, the IPC bindings generator changed very little, and thus, our patches to the generator that create the fuzzer hooks apply cleanly. However, the switch to Mojo IPC was not completed in Chrome 69, leading us to add fuzzer hooks for the remaining legacy interfaces manually. The renderer-side checks also differ significantly between the versions, thus requiring us to add patches that turn off the checks manually.

In addition to the browser patches, we modified the browser instrumentation to handle the old browser versions. Chrome 99 required us to switch to Playwright 1.18.1. No version of Playwright supports Chrome 69. Therefore, we use Puppeteer. Anecdotally, the instrumentation for old browser versions required more effort than backporting the browser patches.

We run the fuzzer for a maximum of 24 hours and measure the time until our fuzzer successfully reproduces and detects the known vulnerabilities from Table II. The first vulnerability to be reproduced is CVE-2019-5856. The fuzzer triggers the bug in less than a minute of runtime. We must turn off the sanitizer for this bug to trigger CVE-2018-18345 because CVE-2019-5856 is triggered so often that it impedes other executions. The fuzzer then triggers CVE-2018-18345 after approximately 14 minutes. The last vulnerability to be reproduced is CVE-2022-1637 after 11.4 hours.

Oracle Evaluation To further evaluate the proposed oracles, we randomly sample and reproduce five SI bypass vulnerabilities from the list of known vulnerabilities. We apply our sanitizer and IPC mutation patches to the affected browser version. We also apply the browser patches provided as proof-of-concept in the bug report. If the vulnerability leaks data from a specific storage (e.g., the clipboard), we seed this storage with the magic string. Next, we execute the PoC and check if one of our oracles logs the SI bypass. We failed to reproduce the proof-of-concept of CVE-2021-21175, so we replaced this testcase with another random sample.

Table III contains the results of our evaluation. Of the five random test cases, the process sanitizer detects one exploit, and the leak sanitizer detects 3 exploits. Both sanitizers fail to detect CVE-2022-3044. This bug allows any compromised renderer to leak clipboard contents without user permission. The proof-of-concept utilizes the MojoJS bindings to send IPC messages by invoking JavaScript methods. Since these bindings utilize different code paths, our leak sanitizer does not observe the leaking message. We note that the sanitizer detects clipboard leaks if the message is sent from the C++ IPC library. Including the three known bugs that were used to evaluate the fuzzer, we observe a false negative rate of 12.5%. Thus, we conclude that the oracles reliably detect different SI bypass bugs.

False Positives The two oracles do not produce false positives due to their design. They utilize the ground truth of the generator (i.e., the site of the generated document) and transfer it to the process level. Given that the victim document behaves benignly, any observed cross-site data flow or process-reuse constitutes a SI bypass. We did not observe false positives during the evaluation or the fuzzing campaign.

C. Code Coverage

We examine code coverage to evaluate the effectiveness of our generator and browser instrumentation in covering a large amount of browser behavior. Due to the large code base, regular source-based coverage produces a significant performance penalty. Instead, we measure coverage based on LLVM sanitizer coverage. We use LLVM trace-pc-guards that write 1 to a bitmap for every covered edge. We place the bitmap in shared memory to collect coverage from all different processes. The total number of edges is high: 6.9 million edges for Chrome and 3.0 million for Firefox.

We compare our fuzzer to Fuzzorigin [16], a fuzzer for UXSS vulnerabilities. Fuzzorigin also creates HTML documents for two different hosts, albeit with different origins, not different sites. Fuzzorigin utilizes Selenium to instrument the browser under test. Since Fuzzorigin's browser instrumentation is incompatible with recent browsers, we use our browser instrumentation to control the browser and only create the input documents with Fuzzorigin.

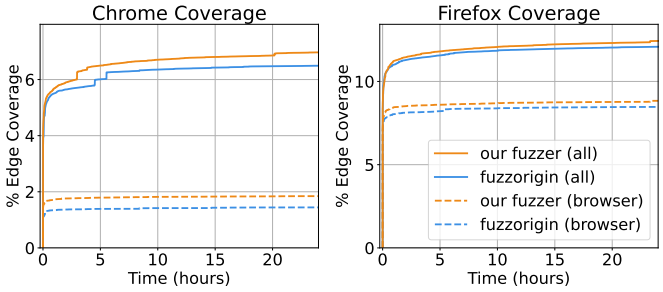

Fig. 7. Edge Coverage over Chrome and Firefox (24 hours)

TABLE IV
SI BYPASS FUZZER FINDINGS

Browser	Description	Class	Severity	ID
	renderer can load arbitrary site	3	S2	CVE-2024-9392
	`Window.name` leaks	1	S4	#384781865
	visited URLs are leaked for styling	1	S3	#1938107
	CORB missing	1	S3	#1532642†

†: The discovered bug is an instance of a known issue, tracked under this ID

We run both fuzzers on Chrome and Firefox with the same resources, that were described at the beginning of this section, and a 24-hour runtime. Neither Fuzzorigin nor our fuzzer utilizes a seed corpus. Figure 7 shows the collected edge coverage in relation to the total number of edges in solid lines. We note that the overall coverage is low. This is expected since we consider large parts of the browser's code base, for example, the HTML parser, out of scope and therefore do not create complex HTML markups.

Coverage measured over all processes might not be meaningful because the vulnerabilities we search for are only in privileged processes like the browser or network process. Complex HTML markups, like the ones generated by Domato [19], cover significant parts of the HTML parser in the renderer but do not lead to many IPC interactions, much less any meaningful behavior of the browser process. Thus, we also measure coverage only over the privileged browser process in a separate bitmap. We configure the browsers not to create separate processes for networking and storage so that coverage over these high-privilege services is included in our collection. The dashed lines in Figure 7 show the coverage collected from the privileged process alone. We note that coverage is still computed with the total number of edges as the denominator because there is no straightforward way to obtain the number of edges of the browser process only.

D. Fuzzing campaign

We run a month-long fuzzing campaign to discover new bugs in recent versions of Chrome and Firefox. We run our fuzzer for two weeks each on patched builds of Chromium 127.0.6497.0 (`6ac2222a`) and Mozilla Firefox 121.0 (`c00a6f0c`) on a system with an AMD EPYC 7702P processor with 128 cores and 500 GB of memory. To fully utilize the available resources, we parallelize our setup, running 50 instances of our fuzzer in Docker containers.

Findings We discovered four security bugs that we list in Table IV. The impact of the bugs ranges from cross-site data leaks to complete control of the victim's website.

- **`Window.name` leak** The `name` property is not reset on cross-site navigations, in non-compliance with the HTML standard [20]. Thus, the browser leaks the name to the next website. This bug was detected by the leak sanitizer. It highlights the relevance of our generated victim page actively seeding various storages by utilizing the magic string in random ineractions with the browser API.
- **leak of visited URLs** The browser process broadcasts all visited links to all renderer processes to facilitate CSS `visited:` styling. Thus, a compromised renderer can sniff all URLs that the victim opens in the browser. The leak sanitizer detected this bug.
- **missing CORB** This was our fastest finding, also detected by the leak sanitizer. Firefox does not implement Cross-Origin Read Blocking, thus leaking the results of no-cors cross-site requests to the renderer. Altough the results are not returned to the JavaScript context, they are visible to the compromised renderer. The Firefox developers are working on a comprehensive implementation of CORB.
- **history origin confusion** A compromised renderer process could force an origin confusion in the browser process using the `history` API and trick the browser process into loading cross-site content in the compromised renderer. Both sanitizers detected this bug. This vulnerability was assigned CVE-2024-9392 and rewarded with an $8,000 bug bounty. We will detail it in the following.

Case Study: Firefox History Confusion A compromised renderer could spoof the URL set via the `history.replaceState` method. While there were checks that verified that the passed URL is of the same origin as the current URL, they were implemented on the renderer side. Our fuzzer-generated input circumvented the check by replacing the passed URL in the IPC message sent to the browser process. The message led to an origin confusion in the browser process, which now stored the spoofed URL as the last URL of this frame. A subsequent reload of the document would trigger the browser process to load the document at the spoofed URL into the compromised renderer process. Thus, the compromised renderer could access the document and cookies of the victim origin and execute JavaScript in the context of the victim origin. Figure 8 shows the simplified proof-of-concept exploit for the vulnerability discovered by the fuzzer. It uses the `IPCFuzzer` API introduced by our browser patches to detect leaks in incoming IPC messages and replace the URL in the outgoing IPC message. This finding confirms that the IPC message mutations in combination with Web IDL driven browser API interactions effectively uncover SI bypass bugs.

E. Quantifying Renderer Kills

We execute our fuzzer for 10 hours on Chrome and Firefox, and count the iterations that result in the Playwright error,

```
IPCFuzzer.activate_leak_sanitizer();
IPCFuzzer.mutate_url("http://127.0.0.2:8080/victim.html");
window.history.replaceState("foo","", null);
window.location.reload();
```

Fig. 8. Proof-of-Concept for Firefox History Confusion

that is caused by the browser process killing a renderer that it observes to act maliciously. For Chrome, 8.7% of iterations ended in such a Playwright error, indicating a killed renderer. For Firefox, the proportion of such Playwright errors increased to 13.1%. Thereby, we observe that even our selected and targeted IPC mutations trigger the security mechanism of the browser process at a high frequency.

VII. Discussion

We start by reexamining our research questions in light of our experimental results.

RQ1 We evaluated two sanitizers for SI bypasses: The leak sanitizer detects data leaked from the victim site to the attacker process by observing the incoming IPC messages. The process sanitizer detects cross-site process reuse or sharing by tagging processes with the first content's site. Both sanitizers infer the correct site from the HTML document.

Our evaluation on known bugs and current versions of browsers reveals that both sanitizers are effective in detecting SI bypass bugs. The process sanitizer only catches very specific bugs and is not applicable to the known vulnerabilities. However, it did detect the history confusion bug in Firefox from our case study. Fundamentally, the process sanitizer covers a blind spot of the leak sanitizer: cross-site process re-use without explicit leaks of secret data cannot be detected by the leak sanitizer.

Our fuzzer successfully reproduces known vulnerabilities and discovers new vulnerabilities in Chrome and Firefox. In doing so, the sanitizers produce no false positives. The attacker process should never receive the secret victim data unless the victim site intentionally transmits the data via cross-site communication APIs. Thereby, the sanitizers constitute an elegant solution to the problem of detecting SI bypasses.

RQ2 We analyzed 39 bug reports to identify the common preconditions to trigger SI bypass vulnerabilities. We identified two main components: spoofing origin parameters of IPC messages and bypassing renderer-side checks. We evaluated a fuzzing approach simulating the compromised renderer by modifying outgoing IPC messages with random mutations.

Our evaluation confirms that this approach is effective in triggering SI bypass bugs. By automating the arbitrary malicious behavior of the renderer process, we can evaluate the behavior of the browser process under realistic attack conditions. The discovered CVE-2024-9392, for example, could not be triggered without both components of malicious behavior.

Including the commands for the IPC fuzzer in the HTML documents produced highly reproducible proofs-of-concept. The discovered bugs can easily be reproduced by manually browsing the input documents with the patched browser.

An unexpected finding of our research was that debug assertions are detrimental to our fuzzer. Debug assertions completely prevented the reproduction of one of the three known bugs and slowed down the reproduction of the other two known bugs by a factor of three.

a) Limitations: The browser process kills renderer processes upon receiving IPC messages that fail the security checks to contain the compromise. These renderer kills limit the performance of our fuzzer. The browser instrumentation libraries throw an error upon loss of connection to the renderer, requiring a costly restart of the whole browser.

We evaluated patching out the renderer-killing behavior of the browser process to combat this performance penalty. However, we find that without renderer kills, our fuzzer regularly discovers false positives (e.g., SI bypasses that cannot be reproduced in a browser with renderer kills). The browser process utilizes the kill function to clean up whenever it detects an inconsistent state. All vulnerabilities following that inconsistent state do not affect the regular browser, but occur in our experiment because we removed the security mechanism.

The performance penalty inflicted by the renderer kills limits our ability to try different manipulations of IPC messages. We tested different mutation strategies early, but the frequency of renderer kills greatly exceeded the one described in Section VI-E, thereby preventing any meaningful execution. Thus, we focus on origin-related manipulations that are the most frequent exploit path for Site Isolation bypasses. Our fuzzer does not create random IPC messages from scratch, a capability that would be required to exploit three prior vulnerabilities that we marked out-of-scope in Table V.

The WebKit developers were in the process of implementing Site Isolation at the time of writing. We validated that support for WebKit can easily be added by applying similar patches as the ones described in Section V on WebKit's IPC layer. However, while Safari supports running custom WebKit artifacts, Safari's Webdriver implementation does not support this. This was confirmed by the WebKit developers. Thus, we cannot instrument and fuzz Safari in a similar way to the other browsers. While we can instrument and fuzz the open-source GTK frontend for WebKit (WebKitGTK), this frontend differs significantly from the closed-source Safari frontend. At the time of writing, WebKitGTK did not support Site Isolation.

b) Reproducibility: Since our fuzzer relies on browser source code patches to mutate IPC messages, reproducing our experiments incurs the overhead of patching and compiling the browser. We provide the full source code of the forked and patched browsers on GitHub, accompanied by Docker containers and documentation to build the old browser revisions, to increase reproducibility. While our patches apply cleanly even on new browser versions, they might require expert knowledge in the future, should the browser codebase change significantly.

c) Future Work: Coverage-guided fuzzing proved to be incredibly useful in finding bugs [21]. However, the impact of coverage guidance in browser fuzzing is the content of discussion [22]. On the other hand, it could guide the manipulations of IPC messages toward messages that pass security checks.

In addition to code coverage, JavaScript execution feedback from exceptions and error messages could guide JavaScript generation towards semantically valid statements.

We did not include extensions in the threat model. Extensions have access to powerful APIs and may be granted access to all sites opened in the browser. By generating malicious extensions as part of the input, our fuzzer could discover vulnerabilities that allow an extension to bypass access checks or even access privileged origins like `chrome://`, leading to sandbox escapes.

VIII. Related Work

Adjacent to our work, Kim et al. [23] examined vulnerabilities in browser extensions that lead to UXSS and SI bypass and Gierlings et al. [24] exploited Site Isolation to facilitate DoS attacks on the host system.

Semantic Browser Bugs Recently, semantic browser bugs received special attention from the research community: Shou et al. [25] implemented a fuzzer for Cross-Origin-Read-Blocking (CORB) bugs and evaluated Chrome's CORB implementation. Most notably, Kim et al. [16] proposed the fuzzer Fuzzorigin to discover Universal Cross-Site Scripting vulnerabilities in browsers. Our approach is similar to theirs; we also process several documents in the browser to implement an oracle on top of the browser state. In contrast to Fuzzorigin, our fuzzer can trigger SI bypass vulnerabilities because it simulates the renderer compromise. Furthermore, our sanitizers detect data leaks into the compromised renderer process that are not visible to Fuzzorigin's origin sanitizer.

Fuzzorigin relies on a manually created grammar for JS code generation. This grammar covers only a limited set of APIs, thus limiting the fuzzers coverage of cross-site navigations and interactions. For example, the `Window.name` property that is relevant for the discovered Chrome bug is not supported by the generator. The same holds true for blob URLs, which were relevant for several previous vulnerabilities. Complemented by our new oracles and IPC message mutation, Fuzzorigin's generator could also have uncovered the other three new vulnerabilities. To maximize coverage of cross-site interactions, we implemented a more general approach that utilizes Web IDL information to achieve complete coverage of the browser's JS API.

Other approaches include formal models to detect semantic bugs during web platform test executions [26] or in the web standard [27]. Wi et al. [28] used differential testing to discover CSP bugs in browsers. They used the other browser's behavior as an implicit oracle. Rautenstrauch et al. [29] proposed discovering cross-site leaks in browsers by automatically checking for leaks across test executions.

Browser Fuzzing Most browser fuzzers targeted the DOM engine [19], [22], [30]–[32] or JavaScript engine's JIT compiler [9]–[11] to discover the security bugs leading to the renderer compromise that we presuppose in our work. Related to our work, Zhou et al. [31] and Wang et al. [32] also leveraged Web IDL interface specifications to generate semantically valid JS. Recent publications specifically targeted browser APIs for GPU-supported rendering [33], [34]. The discovery of memory bugs in the privileged browser process usually involves fuzzing the process's IPC interfaces. Pan et al. [35] fuzzed Chrome's IPC interfaces via Mojo's JavaScript bindings. Yang et al. [36] fuzzed IPC services on macOS to break out of the Safari sandbox. Schumilo et al. [37] proposed snapshot fuzzing to overcome costly browser restarts.

IX. Conclusion

Our research sheds light on a so far unexplored type of vulnerability that allows attackers to bypass the new Site Isolation security mechanism of modern browsers. The key problem of Site Isolation is that the browser must correctly store the site of every renderer process and enforce security checks on all IPC messages sent by the renderers. Based on our analysis of all public reports of SI bypass bugs, we discern three different classes: vulnerabilities caused by missing checks on IPC messages, those caused by invalid checks, and those caused by the privileged browser confusing the site of a renderer. Leveraging this classification and information from the public bug reports, we identify the common preconditions of an SI bypass exploit. In particular, SI bypass exploits require an attacker to have compromised the renderer process to spoof IPC messages and circumvent renderer-side security checks. With these insights, we design and implement a fuzzer to trigger Site Isolation bypass vulnerabilities by simulating the malicious behavior of a compromised renderer. We evaluated process-level and data-flow-based oracles that detect cross-site data leaks and process-sharing, finding that they effectively detect Site Isolation bypass vulnerabilities. We first demonstrated the practicability of this approach by evaluating old browsers with known SI bypass vulnerabilities. Our fuzzer also has proven effective in uncovering new vulnerabilities, as it discovered four security bugs in current versions of Chrome and Firefox.

Acknowledgment

We gratefully acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972 as well as from the European Union's Horizon 2020 research and innovation programme under project TESTABLE, grant agreement No 101019206. We thank Tobias Jost for his technical support and his deep knowledge of C++ and CMake.

Ethics Considerations

We only test browser executables locally and do not interfere with genuine websites. Since the vulnerabilities discovered during our fuzzing campaign might be used to attack users, we confidentially disclose the vulnerabilities via the available channels for security bugs. We support the developers in fixing the bugs and keep our findings secret until the developers make the bug reports public. In doing so, we adhere to the rules and guidelines for reporting security bugs, which the browser developers define.

The discovered bugs were reported no later than December 2024. The bug that received a CVE was fixed within 3 months of the report. The three other bugs are considered less severe and the bug reports or their duplicates were since made public by the developers, although the bugs were not yet fixed. Since the three bugs are public, their description in this publication does not cause harm.

REFERENCES

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution." in *IEEE Security & Privacy*, 2019, pp. 1–19.

[2] C. Reis, A. Moshchuk, and N. Oskov, "Site isolation: Process separation for web sites within the browser." in *USENIX Security Symposium*, 2019, pp. 1661–1678.

[3] Mozilla, "Firefox 95.0 release notes," Mozilla, 2021, visited 2024-07-16. [Online]. Available: https://www.mozilla.org/en-US/firefox/95.0/releasenotes/

[4] T. C. Team, "Addresssanitizer," 2025, visited 2025-12-02. [Online]. Available: https://clang.llvm.org/docs/AddressSanitizer.html

[5] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," in *IEEE Transactions on Software Engineering*, vol. 47, 2021, pp. 2312–2331.

[6] A. Agarwal, S. O'Connell, J. Kim, S. Yehezkel, D. Genkin, E. Ronen, and Y. Yarom, "Spook.js: Attacking chrome strict site isolation via speculative execution." in *IEEE Security & Privacy*, 2022, pp. 699–715.

[7] Chromium, "Mojo," Chromium, 2024, visited 2025-01-02. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/main/mojo/

[8] J. Lim, Y. Jin, M. Alharthi, X. Zhang, J. Jung, R. Gupta, K. Li, D. Jang, and T. Kim, "Sok: On the analysis of web browser security," in *arXiv preprint*, 2021.

[9] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz, "Jit-picking: Differential fuzzing of javascript engines." in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022, pp. 351–364.

[10] S. Groß, S. Koch, L. Bernhard, T. Holz, and M. Johns, "Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities." in *Network and Distributed System Security (NDSS) Symposium*, 2023.

[11] J. Wang, Z. Zhang, S. Liu, X. Du, and J. Chen, "Fuzzjit: Oracle-enhanced fuzzing for javascript engine jit compiler." in *USENIX Security Symposium*, 2023, pp. 1865–1882.

[12] M. Bugtracker, "Create authoritative 'this origin to this content process' infrastructure," Mozilla Bugtracker, 2025, visited 2025-01-07. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1491018

[13] ——, "Enforce content process restrictions in ipc," Mozilla Bugtracker, 2025, visited 2025-01-07. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1484019

[14] ——, "Fission site sandboxing," Mozilla Bugtracker, 2025, visited 2025-01-07. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1505832

[15] N. Bars, M. Schloegel, T. Scharnowski, N. Schiller, and T. Holz, "Fuzztruction: Using fault injection-based fuzzing to leverage implicit domain knowledge." in *USENIX Security Symposium*, 2023, pp. 1847–1864.

[16] S. Kim, Y. M. Kim, J. Hur, S. Song, G. Lee, and B. Lee, "Fuzzorigin: Detecting uxss vulnerabilities in browsers through origin fuzzing." in *USENIX Security Symposium*, 2022, pp. 1008–1023.

[17] Chromium, "Check, dcheck and notreached," 2025, visited 2025-01-21. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/HEAD/styleguide/c++/checks.md

[18] F. S. Docs, "Firefox source docs: Fuzzing," 2025, visited 2025-01-21. [Online]. Available: https://firefox-source-docs.mozilla.org/tools/fuzzing/index.html

[19] I. Fratric, "Domato," Google Project Zero, 2024, visited 2025-01-02. [Online]. Available: https://github.com/googleprojectzero/domato

[20] WHATWG, "Html living standard," 2025, visited 2025-01-20. [Online]. Available: https://html.spec.whatwg.org/multipage/browsing-the-web.html#resetBCName

[21] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[22] W. Xu, S. Park, and T. Kim, "Freedom: Engineering a state-of-the-art dom fuzzer." in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020, pp. 971–986.

[23] Y. M. Kim and B. Lee, "Extending a hand to attackers: Browser privilege escalation attacks via extensions." in *USENIX Security Symposium*, 2023, pp. 7055–7071.

[24] M. Gierlings, M. Brinkmann, and J. Schwenk, "Isolated and exhausted: Attacking operating systems via site isolation in the browser." in *USENIX Security Symposium*, 2023, pp. 7037–7054.

[25] C. Shou, İ. B. Kadron, Q. Su, and T. Bultan, "Corbfuzz: Checking browser security policies with fuzzing," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 215–226.

[26] P. Bernardo, L. Veronese, V. D. Valle, S. Calzavara, M. Squarcina, P. Adão, and M. Maffei, "Web platform threats: Automated detection of web security issues with wpt." in *USENIX Security Symposium*, 2024.

[27] L. Veronese, B. Farinier, P. Bernardo, M. Tempesta, M. Squarcina, and M. Maffei, "Webspec: Towards machine-checked analysis of browser security mechanisms." in *IEEE Security & Privacy*, 2023, pp. 2761–2779.

[28] S. Wi, T. T. Nguyen, J. Kim, B. Stock, and S. Son, "Diffcsp: Finding browser bugs in content security policy enforcement through differential testing." in *Network and Distributed System Security (NDSS) Symposium*, 2023.

[29] J. Rautenstrauch, G. Pellegrino, and B. Stock, "The leaky web: Automated discovery of cross-site information leaks in browsers and the web." in *IEEE Security & Privacy*, 2023, pp. 2744–2760.

[30] C. Zhou, Q. Zhang, M. Wang, L. Guo, J. Liang, Z. Liu, M. Payer, and Y. Jiang, "Minerva: browser api fuzzing with dynamic mod-ref analysis," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, p. 1135–1147.

[31] C. Zhou, Q. Zhang, L. Guo, M. Wang, Y. Jiang, Q. Liao, Z. Wu, S. Li, and B. Gu, "Towards better semantics exploration for browser fuzzing," in *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2023.

[32] J. Wang, P. Qian, X. Huang, X. Ying, Y. Chen, S. Ji, J. Chen, J. Xie, and L. Liu, "Tacoma: Enhanced browser fuzzing with fine-grained semantic alignment," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, p. 1174–1185.

[33] H. Peng, Z. Yao, A. A. Sani, D. Tian, and M. Payer, "Gleefuzz: Fuzzing webgl through error message guided mutation." in *USENIX Security Symposium*, 2023, pp. 1883–1899.

[34] L. Bernhard, N. Schiller, M. Schloegel, N. Bars, and T. Holz, "Darthshader: Fuzzing webgpu shader translators & compilers." in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024, pp. 690–704.

[35] G. Pan, T. Luo, Y. Tao, X. Lei, S. Chen, H. Liu, and C. Wu, "Amf: Efficient browser interprocess communication fuzzing," in *2023 20th Annual International Conference on Privacy, Security and Trust (PST)*, 2023, pp. 1–6.

[36] K. Yang, H. Zhao, C. Zhang, J. Zhuge, and H. Duan, "Fuzzing ipc with knowledge inference," in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, 2019, pp. 11–1109.

[37] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-net: network fuzzing with incremental snapshots," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, p. 166–180.

APPENDIX A
KNOWN BUGS

TABLE V
SITE ISOLATION BYPASS VULNERABILITIES IN CHROME SINCE 2018 AND FIREFOX SINCE 2021

Browser	ID	Description	Class	In Scope
C	CVE-2024-1671	Origin confusion in session history leaks URL of srcdoc iframe	3	●
F	CVE-2024-0748	Compromised renderer can set arbitrary document URI	1	●
C	CVE-2022-4913	Compromised renderer can access extension storage	1	◑
C	CVE-2022-3661	Compromised renderer can message any extension content script	1	◑
C	CVE-2022-3044	No access checks for clipboard interface	1	●
C	CVE-2022-1637	Cross-origin iframe can spoof the hostname of top-frame by opening new window with `javascript:` URI and target `_blank`	3	●
C	CVE-2022-0305	Hidden bug report for Service Worker	?	?
C	CVE-2022-0294	No checks in PushMessaging interface that verify if the referenced ServiceWorker belongs to the same origin as the renderer	1	●
C	CVE-2022-0292	Fenced frame can open `file:` URLs	1	●
C	CVE-2022-0291	Hidden bug report for storage	?	?
F	#827853	Compromise renderer can inject HTTP headers	1	○
F	#1770227	Compromised renderer can forge notifications	1	○
C	#40060671	Compromised renderer can spoof PortContext and claim to be WorkerContext of arbitrary extension	1	◑
C	CVE-2021-38010	`URLLoader` leaked to ServiceWorker, compromised renderer can read the response of redirected cross-origin requests	1	○
C	CVE-2021-30507	Compromised renderer can spoof `X-Chrome-offline` header to read arbitrary file	1	○
C	CVE-2021-21222	TOCTOU bug in `GeneratedCodeCache`: compromised renderer can change value after the hash computation	2	○
C	CVE-2021-21175	`X-Frame-Options` error of cross-origin iframe is leaked to parent	1	●
C	#40054801	Compromised renderer that outlives state in the browser process can bypass security checks to spoof origin	2	●
F	#1713203	Cookies leaked to all processes	1	●
C	CVE-2020-6435	Compromised renderer can spoof sender id to extension	1	◑
C	CVE-2020-6385	Origin checks in `BlobURLStoreImpl::Register` skipped if renderer process simulates detachment	2	●
C	CVE-2020-6380	Compromised renderer can spoof origin, message any extension	1	◑
C	CVE-2019-13763	Compromised renderer can spoof origin and leak data from `PaymentManager`	1	●
C	CVE-2019-13738	Sandboxed iframe shares execution context with initial non-sandboxed `about:blank` frame	3	●
C	CVE-2019-13727	Compromised renderer can create WebSocket to arbitrary URL and leak the response headers	1	●
C	CVE-2019-13682	Spoofing origin in protocol handler registration leads to SI bypass	1	●
C	CVE-2019-5865	CORS bypass: compromised renderer can set `Host` header during redirect	1	○
C	CVE-2019-5862	Compromised renderer can spoof `document_url_` and register arbitrary files from victims site in AppCache	1	●
C	CVE-2019-5856	Missing browser-side checks, compromised renderer can access filesystem of other origins	1	●
C	CVE-2019-5773	Compromised renderer can spoof origin when accessing IndexedDB	1	●
C	#40093845	Compromised renderer can spoof origin and access code cache of other site	1	●
C	#40093844	Invalid checks on `ws:` URLs, compromised renderer can leak cookies	2	●
C	CVE-2018-18345	`BlobURLRegistry::RegisterURL` access check based on renderer provided `host` and `public_url`	1	●
C	CVE-2018-16074	BlobURLs created from different opaque origins have opaque origin but are all handled in the same process	3	●
C	CVE-2018-16073	Data URL in iframe is loaded in same process if embedding page is loaded from cache	3	●
C	CVE-2018-6165	Refresh during navigation triggers origin confusion	3	●
C	CVE-2018-6121	Compromised renderer can commit url of extension	1	●
C	#40092826	Cookies leaked to cross-site renderer in presence of DevTools	1	●
C	#40092525	Compromised renderer can spoof origin during filesystem url creation	1	●

All known Site Isolation bypass vulnerabilities in Chrome and Firefox, filtered from all bugs in the Chromium bug tracker with the tag *Internals>Sandbox>SiteIsolation*, Firefox meta-bug-trackers [12]–[14], and NVD entries whose description included the term *Site Isolation*
●: bug is in scope of our work; ◑: in scope but requires extensions as fuzzer input; ○: out of scope; ?: unknown
1: Checks missing; 2: Checks bypassed; 3: Origin confusion

APPENDIX B
GENERATED INPUT EXAMPLE

```
<html>
    <head>
        <script>
            IPCFuzzer.deactivate_renderer_checks(); IPCFuzzer.activate_leak_sanitizer();
        </script>
        <script>
            navigator.serviceWorker.register('/sw.js').then((reg) => {reg.update();}).catch((e) => {});
        </script>
        <script>
            try {
                function check_site_violation(src, exec, token){
                    let fetch_url = new URL(src);
                    let exec_url = new URL(exec);
                    if(fetch_url.hostname != exec_url.hostname){
                        console.log('[UXSS] [exec-' + token + '] ' + fetch_url.hostname + ' ' + exec_url.hostname);
                        return true;
                    }
                    if(IPCFuzzer.check_isolation(src)){
                      console.log('[UXSS] [process-' + token + '] ' + src);
                      return true;
                    }
                    try {console.log(document.cookie);} catch {}
                    try {cookieStore.get('sanitize').then(c => console.log(c)} catch {}
                    try {console.log(localStorage.getItem('sanitize'))} catch {}
                    // ... IndexedDB, OPFS, fetch
                    return false;
                }
            } catch {}
            // ... similar sanitizer for iframes and window objects
        </script>
    </head>
    <body>
        <script>
            async function foo() {
                // try-catch-finally blocks around each statement omitted for clarity
                check_site_violation('http://127.0.0.1:8080', location.origin, '1c26c410');

                IPCFuzzer.mutate_url('http://127.0.0.1:8080/input-14094_page-1.html#549d489c');
                var var0 = 'foo';
                var var1 = {fatal: window.closed, };
                var var2 = new TextDecoder(var0,var1);
                var var3 = {stream: window.closed, };
                var var4 = await var2.decode('',var3);
                var var6 = 'no-referrer-when-downgrade';
                var var7 = 'same-origin';
                var var8 = 'include';
                var var9 = 'force-cache';
                var var10 = 'follow';
                var var11 = 'auto';
                var var12 = 'half';
                var var13 = 'local';
                var var15 = '1';
                var var16 = 'token-redemption';
                var var17 = 'refresh';
                var var14 = {version: var15, operation: var16, refreshPolicy: var17, };
                var var18 = {eventSourceEligible: window.closed, triggerEligible: window.closed, };
                var var5 = {method: var0, headers: '', body: var0, referrer: var0, referrerPolicy: var6, mode: var7,
                credentials: var8, cache: var9, redirect: var10, integrity: var0, keepalive: window.closed, priority: var11,
                browsingTopics: window.closed, adAuctionHeaders: window.closed, sharedStorageWritable: window.closed,
                duplex: var12, targetAddressSpace: var13, privateToken: var14, attributionReporting: var18, };
                var var19 = new Request(var0,var5);
                var var20 = await var19.formData();
                IPCFuzzer.mutate_site_for_cookies_replace_host('http://127.0.0.2:8080/input-14094_page-2.html');
                var var21 = document.createElement('param');
                var var22 = document.createElement('object');
                document.body.appendChild(var22);
                var22.appendChild(var21);
                // ...
            }
            foo();
        </script>
    </body>
</html>
```

Fig. 9. Example input produced by the generator

APPENDIX C
ARTIFACT APPENDIX

A. Description & Requirements

The repository contains the source code of the fuzzer described in the paper. It is a browser IPC fuzzer to discover site isolation bypass vulnerabilities. The fuzzer utilizes WebIDL definitions to generate HTML/JS inputs utilizing the browser JS API. The browser is instrumented with Playwright to simulate user interactions. We patched Chrome and Firefox to add our Site Isolation bypass bug oracles and the IPC fuzzer component that mutates IPC messages sent by the renderer process. The patched browsers are located in other repositories of the same GitHub organization.

The artifact contains all the code to build and run the fuzzer, reproducing the fuzzing campaign and the experiments (e.g., evaluating coverage and bug finding capabilities) of the paper.

1) How to access: The artifacts are located at https://github.com/si-bypass-fuzzing. The fuzzer and the repositories holding the patched browsers are located in this GitHub organization and linked in the README. An immutable version of the fuzzer repository is located at https://doi.org/10.5281/zenodo.17750615. The `patches` directory contains the browser patches in diff format.

2) Hardware dependencies: The following system requirements apply to building the patched browser versions required to run the fuzzer:

- an x86-64 machine
- at least 16GB RAM
- 200 GB disk space

3) Software dependencies:

- Ubuntu 22.04
- Python3.12
- git
- Docker
- tmuxp
- to build the current Chrome without a Docker container, the Chrome build dependencies must be installed

4) Benchmarks: None