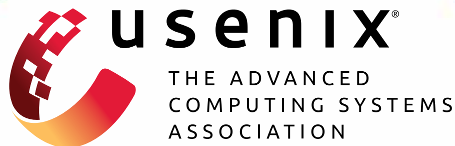


FuzzOrigin: Detecting UXSS vulnerabilities in Browsers through Origin Fuzzing

Sunwoo Kim, *Samsung Research;* Young Min Kim, Jaewon Hur, and Suhwan Song, *Seoul National University;* Gwangmu Lee, *EPFL;* Byoungyoung Lee, *Seoul National University*

https://www.usenix.org/conference/usenixsecurity22/presentation/kim

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.

FuzzOrigin: Detecting UXSS vulnerabilities in Browsers through Origin Fuzzing

Sunwoo Kim*
Samsung Research
sunwoo28.kim@samsung.com

Young Min Kim
Seoul National University
ym.kim@snu.ac.kr

Jaewon Hur
Seoul National University
hurjaewon@snu.ac.kr

Suhwan Song
Seoul National University
sshkeb96@snu.ac.kr

Gwangmu Lee†
EPFL
gwangmu.lee@epfl.ch

Byoungyoung Lee‡
Seoul National University
byoungyoung@snu.ac.kr

Abstract

Universal cross-site scripting (UXSS) is a browser vulnerability, making a vulnerable browser execute an attacker's script on any web pages loaded by the browser. UXSS is considered a far more severe vulnerability than well-studied cross-site scripting (XSS). This is because the impact of UXSS is not limited to a web application, but it impacts each and every web application as long as a victim user runs a vulnerable browser. We find that UXSS vulnerabilities are difficult to find, especially through fuzzing, for the following two reasons. First, it is challenging to detect UXSS because it is a semantic vulnerability. In order to detect UXSS, one needs to understand the complex interaction semantics between web pages. Second, it is difficult to generate HTML inputs that trigger UXSS since one needs to drive the browser to perform complex interactions and navigations.

This paper proposes FuzzOrigin, a browser fuzzer designed to detect UXSS vulnerabilities. FuzzOrigin addresses the above two challenges by (i) designing an origin sanitizer with a static origin tagging mechanism and (ii) prioritizing origin-update operations through generating chained-navigation operations handling dedicated events. We implemented FuzzOrigin, which works with most modern browsers, including Chrome, Firefox, Edge, and Safari. During the evaluation, FuzzOrigin discovered four previously unknown UXSS vulnerabilities, one in Chrome and three in Firefox, all of which have been confirmed by the vendors. FuzzOrigin is responsible for finding one out of two UXSS vulnerabilities in Chrome reported in 2021 and all three in Firefox, highlighting its strong effectiveness in finding new UXSS vulnerabilities.

*The work is done at Seoul National University as an academic exchange program.
†The work is done while the author is a graduate student at Seoul National University.
‡Corresponding author.

1 Introduction

Modern web browsers feature client-side scripting, enabling highly interactive dynamic web pages. By allowing the script code such as JavaScript [39] or WebAssembly [43] to be executed on the client-side, developers can make a web application powerful like a native app, significantly enriching user experience. From the security perspective, however, client-side scripting may expose a challenging attack surface since a script from an attacker can also be executed. This is particularly crucial considering a typical web application architecture—it often involves multiple players (e.g., a main host server, a media provider, an advertiser, etc.), and a single web page is rendered through complex interactions or navigations among these players. Hence, it is important for browsers to faithfully determine if a given script is not from an attacker, and thus it is safe to execute.

Cross-site scripting (XSS) is one of the extensively studied vulnerabilities [3, 20, 25–28, 30, 44, 45, 49, 55, 56, 58–62], exploiting the issue of client-side scripting. It is a security vulnerability in web applications, which allows attackers to inject client-side scripts into a vulnerable web page. Then the attacker's script is executed on behalf of the victim, thereby stealing security-critical resources (e.g., a session cookie of the vulnerable web application). It is arguably the most common and well-known vulnerability. Popular websites such as Twitter and Facebook had numerous XSS vulnerabilities in the past [23, 46, 53], jeopardizing users' data.

Universal cross-site scripting (UXSS) [31] is similar to XSS, because it scripts across sites—i.e., it allows an attacker to inject and execute code on web pages loaded by the browser. However, the key difference is that UXSS is a vulnerability of web browsers, not web applications. Therefore, it is considered a far more severe vulnerability than XSS. More specifically, the impact of UXSS is *universal*, i.e., it is not limited to a particular web application but rather affects all web applications as long as a victim runs a vulnerable web browser. Should it be found in a browser, it allows an attacker to launch attacks against any website, irrespective of the fact

that such websites alone do not have any security issues.

1.1 Research Challenges

Despite the pressing security needs, UXSS is a relatively unexplored research topic compared to XSS. Focusing on the research direction towards fuzz testing for UXSS (which is the main focus of this paper), we think this is due to the following two unique challenges that UXSS bears to meet the key requirements to perform fuzz testing.

Challenge #1: UXSS Detection. First, it is challenging to generalize a UXSS detection mechanism for fuzzing, because it is essentially a semantic vulnerability. This is important because the key requirement for fuzzing is a vulnerability detection mechanism without false positives. In the case of XSS, the detection is straightforward—one only needs to check if the attacker-provided script is executed. If it is executed, then one can determine it triggered an XSS vulnerability. However, in the case of UXSS, simply having the attacker-provided script being executed does not lead to UXSS. Instead, one must carefully inspect the capability (i.e., origin [65]) that the script execution has been granted. To be specific, one needs to confirm that the attacker's script has higher privileges than it supposed to (i.e., violating the same-origin policy [42]). Since the capability of the attacker's script depends on how a browsed web page interacts with different players (or servers), understanding the interaction semantics is crucial to detect UXSS.

Challenge #2: UXSS Triggering. Second, it is challenging to construct HTML inputs triggering UXSS vulnerabilities for fuzzing. We find that the root cause of UXSS stems from the cases that the capability of the script execution is incorrectly updated due to the semantic mistakes in the browser. Hence, to increase the chance to trigger UXSS, a generated HTML should drive complex interactions between multiple servers (i.e., complex cross-origin page loading), which leads to frequent capability updates with respect to scripts.

1.2 FUZZORIGIN: The First UXSS Fuzzer

This paper presents FUZZORIGIN, a browser fuzzer to detect UXSS vulnerabilities. To the best of our knowledge, FUZZORIGIN is the first UXSS fuzzer. Similar to traditional browser fuzzers [12, 32–34, 67, 69], FUZZORIGIN generates the HTML document embedding JavaScript, based on the knowledge of the language syntax (i.e., grammar awareness of HTML and JavaScript). Then FUZZORIGIN runs a web browser while providing the HTML document in hopes the run triggers UXSS.

Unlike traditional browser fuzzers, FUZZORIGIN designs following two unique features to address aforementioned challenges of UXSS: i) an origin sanitizer to detect UXSS; and ii) prioritizing origin-update operations in generating HTML inputs.

Solution #1: Origin Sanitizer. First, the origin sanitizer of FUZZORIGIN keeps track of server interaction semantics through static origin tagging, which is automatically instrumented into the scripts marking where the script was fetched. Leveraging the static origin tagging, when the script is executed by the browser, FUZZORIGIN is capable of checking if the to-be-executed script is granted with the correct capability (i.e., a correct origin). By design, the origin sanitizer does not have any false positive in detecting UXSS, because the static tagging mechanism is precise.

Solution #2: Prioritizing Origin-update Operations. Second, FUZZORIGIN prioritize origin-update operations in generating HTML inputs. This is based on the observation that the root cause of UXSS vulnerabilities is due to incorrect origin update handling in browsers. To this end, FUZZORIGIN generates HTML inputs triggering complex and interactive navigation operations, which makes the browser perform more frequent origin-update operations. In particular, HTML inputs generated by FUZZORIGIN can be characterized by their complex cross-origin navigation behaviors, where each navigation is chained with another navigation using event handlers.

Implementation and Results. We implemented FUZZORIGIN, which works with most of modern web browsers, including Chrome, Firefox, Edge, and Safari. According to our evaluation, the origin sanitizer of FUZZORIGIN showed no false positives in identifying UXSS vulnerabilities. Over the six months of lengthy, extensive evaluations, if the origin sanitizer reports a potential UXSS vulnerability, it is always confirmed to be true by the respective vendors. FUZZORIGIN's HTML generation with chained-navigation operations indeed raised more frequent origin-updates, allowing FUZZORIGIN to effectively test UXSS-relevant logic in the browser.

Importantly, during the evaluation FUZZORIGIN discovered four new UXSS vulnerabilities (one in Chrome and three in Firefox), which is all confirmed by the respective vendors. We highlight that UXSS vulnerabilities are extremely rare vulnerabilities. In 2021, only two and three UXSS vulnerabilities were confirmed in Chrome and Firefox, respectively, meaning that FUZZORIGIN identified 50% (in Chrome) and 100% (in Firefox) of those.

To summarize, this paper makes the following contributions:

- **Analysis: Demystifying UXSS.** We analyzed two previous UXSS vulnerabilities to demystify challenges from the perspective of fuzz testing.
- **Design: The first UXSS Fuzzer.** We proposed FUZZORIGIN, a UXSS fuzzing framework. It features two unique designs for UXSS: (i) an origin sanitizer to detect UXSS and (ii) origin-update prioritization when generating HTML inputs.
- **Result: New UXSS vulnerabilities.** We found four new UXSS vulnerabilities using FUZZORIGIN, which attributes

50% (in Chrome) and 100% (in Firefox) of all confirmed UXSS vulnerabilities in 2021.

2 Background

This section provides the necessary background to understand FuzzOrigin. We first describe the role of origin in browser security as well as XSS and UXSS vulnerabilities related to the origin (§2.1). Then we describe how browsers keep track of origin within their internal data structure, the DOM tree (§2.2).

2.1 Origin in Browser Security

The Same-Origin Policy. The same-origin policy constitutes a fundamental security mechanism in modern web browsers [66], which strictly defines boundaries between web pages. If two web pages have the same origin, one page can access other page's resources and data without restriction, such as DOM, cookie, `fetch`, `localStorage`, `IndexedDB`, `SharedWorker`, and `BroadcastChannel`. For instance, this policy allows a script embedded in `https://bank.com/list` to access a session cookie stored by another page `https://bank.com/login` as they have the same origin.

An origin is defined as a tuple of (`scheme`, `host`, `port`)[1]. Suppose a web page is located in `http://example.org:8080/page.html`, then it has the origin, (`http`, `example.org`, `8080`). This origin is the same as the origin of `http://example.org:8080/sub.html`. However, it is different from the origin of `https://example.org:8080` (different scheme), `http://example.net:8080` (different host), and `http://example.org:8888` (different port).

Cross-Site Scripting. Cross-site scripting (XSS) is a security vulnerability in web applications, allowing attackers to inject scripts into a vulnerable web page browsed by other users. Exploiting XSS, the attacker's injected script (e.g., JavaScript) is executed on the client-side in the context of the vulnerable web page. This essentially elevates the attacker's privilege to access security-sensitive resources of the vulnerable web page (e.g., a session cookie) and perform actions on behalf of the user. XSS is mainly caused by a lack of proper validation over attacker-provided inputs. For instance, if the application fails to filter out script tags, the attacker may provide a script tag as the input to be included in the vulnerable web page.

XSS is arguably the most common publicly reported security vulnerability. Popular websites such as Twitter and Facebook had XSS vulnerabilities in the past [23, 46, 53], exposing numerous users' security-sensitive data to be exploited. In order to launch XSS attacks against a certain web application, the attacker has to find an XSS vulnerability in the very application. In other words, an XSS vulnerability is specific to a web application, and thus it cannot be used to attack any other web application.

[1]The HTML standard defines the origin as a 4-tuple (i.e., `scheme`, `host`, `port`, and `domain`), but we omit `domain` as it does not change the overall story of this paper.

Universal Cross-Site Scripting. On the other hand, universal cross-site scripting (UXSS) is a vulnerability in web browsers or their plugins, allowing attackers to run their code on behalf of the web page loaded by the web browser. It is similar to XSS, as it creates an XSS condition—i.e., UXSS allows an attacker to execute attacker-injected code on web pages loaded by the browser. However, the difference is that UXSS is *universal*, meaning it is not specific to a particular web application. Since the UXSS vulnerability is in the browser, the attack can be launched against any web page loaded by the browser, including internal pages such as the settings page. Thus, UXSS vulnerabilities are considered the most critical security threat in the web ecosystem. Once found in a major browser, it allows an attacker to launch attacks against any website, irrespective of the fact that such websites alone do not have any security issues.

More importantly, UXSS attacks often have more critical security impacts than typical memory corruption or remote code execution vulnerabilities in modern web browsers. In response to memory corruption attacks and side-channel attacks such as Spectre [22], modern browsers started to employ multi-process architecture [1, 16, 51] and site isolation [13, 52]. Thus, each renderer process is tied to an origin, and the access to other origin's data is prevented by the process isolation. As a result, even if a memory corruption vulnerability in the renderer is exploited, an attacker would not have access to other origin's data. On the contrary, UXSS attacks offer a unique and strong attack vector, as it allows the attacker to access other origin's data.

2.2 Origin Tracking in Browsers

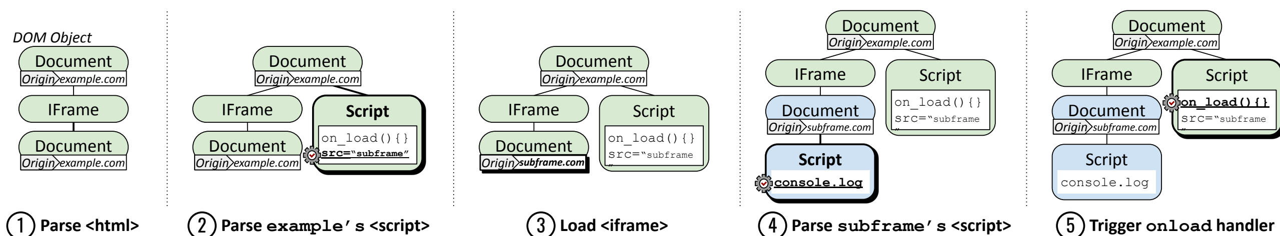
Document Object Tree (DOM) and Origin. In order to enrich the user experience, modern web browsers support client-side scripting (such as JavaScript). In response to an event (i.e., when the browser parses `<script>` tag, when the browser completes the page load, when the keyboard or mouse input is received, when a certain time has elapsed, etc.), a web page can be modified dynamically by executing the client-side script. From the perspective of a browser implementation, the client-side scripting is being supported by the interaction between the renderer and the JavaScript engine. First, the renderer takes web resources (e.g., raw HTML documents) and constructs a document object model (DOM) tree, a logical tree representing the HTML. Then upon a certain event is dispatched, the renderer invokes the JavaScript engine. The JavaScript engine takes the DOM tree from the renderer, and executes the script block corresponding to the dispatched event, modifying the DOM tree. As numerous events are fired throughout loading the web page, frequent interactions

```html
<html>
  <body onload=on_load()>
    <iframe src=""></iframe>
    <script>
      function on_load() {
        // Printing the cookie of http://example.com
        console.log(document.cookie)
      }
      document.querySelector("iframe").src = "http://subframe.com"
    </script>
  </body>
</html>
```

(a) HTML served by `http://example.com`

```html
<!-- embedded in http://example.com's iframe -->
<html>
  <body>
    <script>
      // Printing the cookie of http://subframe.com
      console.log(document.cookie)
    </script>
  </body>
</html>
```

(b) HTML served by `http://subframe.com`



(c) Origin and DOM tree updates by a browser

Figure 1: An example of origin changes in the DOM tree

between the renderer and the JavaScript engine can occur. Thus, the DOM tree is also accordingly kept being updated.

To enforce the same-origin policy, the browser keeps track of the origin as it constructs the DOM tree. Therefore, when the browser executes the JavaScript code triggered by a certain event, it assures that the correct origin is provided. As this origin tracking process is vital in understanding this paper but complex, we take the following simplified example, showing how the browser constructs the DOM tree and embeds the origin for a given HTML document.

Terminology for Describing the DOM Tree. In order to easily describe how the browser internally maintains the DOM tree as well as the associated origin, we denote `document` to be the root element of the DOM tree[2]. `document` conceptually corresponds to the `<html>` tag, and it has an additional property, `origin`, which stores the origin of the `document` and thus represents the context of JavaScript execution within the `document`. When explaining the DOM tree, we intentionally ignore all the HTML tags except `<iframe>` and `<script>` tags, as they are necessary to understand the origin mechanism in browsers.

Example: The Life-Cycle of Origin. In this example, we use two HTML documents; one is fetched from `http://example.com` and another from `http://subframe.com`, where the former loads the latter in its iframe (shown in Figure 1a and Figure 1b). `http://subframe.com` denotes a third-party site, that `http://example.com` may not have control of.

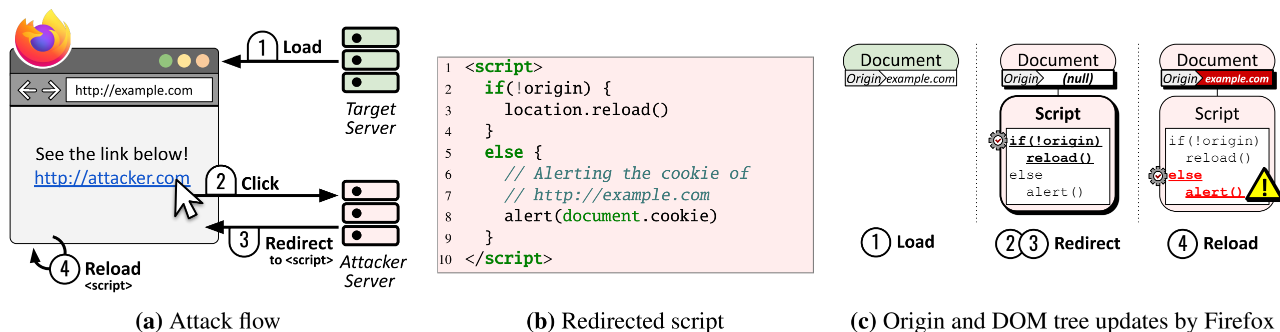
[2]In real-world browser implementations, `window` is the top interface and it has `document` and origin as its property. However, we regard them as the same entity for simplicity, as they have a one-to-one correspondence in most cases.

Once the browser fetches the HTML document from `http://example.com`, it starts parsing it to construct the DOM tree (Figure 1c-①) . The root element is `document` (i.e., `<html>` tag), where its `origin` is initialized to `http://example.com`. The `document` element has the `iframe` element (i.e., `<iframe>` tag) as a child, and `iframe` is initialized to have another `document` as a child. This child `document`'s `origin` is initialized to `http://example.com`, as the HTML standard dictates that an origin of an empty iframe's document inherits the parent document's origin[3].

Next, when the renderer parses the `<script>` tag in HTML (Figure 1c-②), it adds the `script` element to the DOM tree and invokes the JavaScript engine to execute the code in the `script` element. When executing, the JavaScript engine obtains the origin for the script by traversing upward from the `script` element until locating any `document`, which is the origin of the root document (i.e., `http://example.com`). The `script`'s execution sets the `iframe` source to `http://subframe.com`, which updates the child `document` in the DOM tree. This changes the origin of the child `document` to `http://subframe.com` (Figure 1c-③).

Now the renderer starts parsing the new document fetched from `http://subframe.com`, and it inserts another `script` element to the `iframe`'s `document` (Figure 1c-④). In turn, the renderer invokes the JavaScript engine to execute the corresponding `script` (line 6 on Figure 1b), which prints the document's cookie. Here, the origin is specified as `http://subframe.com` (referring to the origin of the `iframe`'s `document`), the cookie of `http://subframe.com` is printed.

[3]It is worth noting that the HTML standard sets special rules in determining the origin of the iframe (e.g., the `sandbox` attribute [38]), but we do not consider them for simplicity.



(a) Attack flow **(b)** Redirected script **(c)** Origin and DOM tree updates by Firefox

Figure 2: A UXSS vulnerability in Firefox due to incorrect origins for data URLs (CVE-2017-5466).

After that, as the page loading is completed (Figure 1c-⑤), the script function `on_load()` (line 5 on Figure 1a) is invoked as it is registered as the `onload` handler. This function prints the cookie of `http://example.com`, because the origin is provided to be the origin of the root `document`.

3 Case Study on Previous UXSS Vulnerabilities

This section analyzes two previous UXSS vulnerabilities in the two most-used browsers, Firefox and Chrome, respectively. Through this analysis, we attempt to showcase how UXSS vulnerabilities occur and how the origin is related.

3.1 Incorrect Origin for Data URL

Origin of Data URL. The `Location` header is an HTTP response header, which redirects a current page to the specified URL [40] if served with a 3xx redirection response.

One URL is a data URL (`data:`), which embeds the data within the URL. If the browser receives a data URL as a redirection target, it loads the embedded data directly. One unique aspect of this data URL is that the origin is `null` (i.e., an opaque origin [65]) per the HTML standard, implying that a data URL page has no origin and thus has no capability to access other pages' resources.

CVE-2017-5466 in Firefox. The root cause of CVE-2017-5466 [36] is that Firefox incorrectly updates the origin if the page is redirected to a data URL and reloaded. Normally, even if the data URL page is reloaded, it should have a `null` origin. However, Firefox incorrectly updates the origin of the reloaded data URL page to the origin of the document before loading the data URL page. As a result, an attacker can execute their malicious JavaScript code on behalf of the origin before the redirection.

The attack can be performed in the following steps as illustrated in Figure 2:

- ① A user navigates to the target page (i.e., `http://example.com`), where the page has a link to the attacker-controlled page (i.e., `http://attacker.com`).
- ② After the target page is loaded in the browser, a user clicks a link to navigate to the attacker's page. Then the browser would request a page from the attacker's server.
- ③ Upon receiving the request, the attacker server responds with the `Location` header pointing to a data URL, which embeds the HTML as shown in Figure 2b. Then the browser performs the in-place redirection to the data URL, which would execute the code in Figure 2b.
- ④ When Firefox executes the JavaScript embedded in the data URL, the `origin` is `null`. As a result, the browser reloads the current page (line 3 in Figure 2b).

The problem occurs at ④, in which the origin should still be `null` after reloading. However, the data URL page is incorrectly updated to the page's origin before the initial redirection (i.e., `http://example.com`). Thus, when the script is executed again after reloading, the attacker's script would be executed on behalf of the target's origin, allowing the attacker's code to access the target's cookie values illegally (line 8 in Figure 2b). Firefox patched this UXSS vulnerability to update to the correct `null` origin when reloading data URL pages.

3.2 Incorrect Origin for Unloaded Document

Origin of Unloaded Document. Upon handling various navigation requests, a browser keeps unloading old documents and loading new documents. As such, the origin of old and new documents should accordingly be updated—i.e., the old (and new) document should have the origin of where the old (and new) document is fetched. From the DOM tree's perspective, such loading and unloading would keep updating references (i.e., updating edges), which may result in a dangling sub-tree. The origin of a dangling sub-tree should be invalidated and no longer updated.

CVE-2015-1293 in Chrome. The vulnerability CVE-2015-1293 [8] occurs as Chrome references an incorrect origin for an unloaded frame document. Due to this vulnerability, if a victim user visits the attacker's page with the target site embedded in an iframe, the attacker's script can be executed on behalf of the target's origin (Figure 3a).

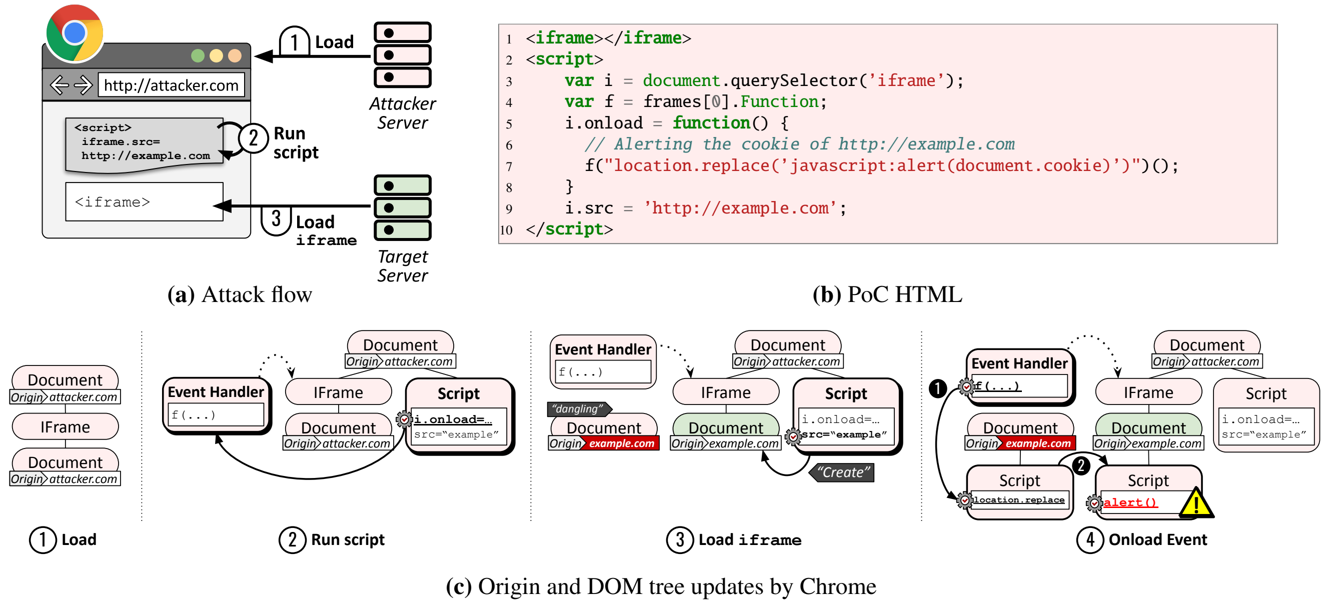


Figure 3: A UXSS vulnerability in Chrome due to incorrect origins for unloaded documents (CVE-2015-1293).

To demonstrate, an example of the attacker's HTML and how the DOM tree is updated is shown in Figure 3b and Figure 3c, respectively.

- ① Upon receiving the attacker's HTML, the browser creates the root `document` and an empty `document` as a child of the `iframe` element. The origin of the root `document` is `http://attacker.com`, and the child `document` inherits the origin of the root according to the HTML standard [64].
- ② The `script` tag is executed, which registers a `load` event handler (line 3 to 8). This event handler is appended to the `iframe`.
- ③ The script execution continues (line 9), which changes the source of the `iframe` to `http://example.com` (line 10). As a result, the `iframe`'s `document` is replaced with the new `document` fetched from `http://example.com`, whose origin is `http://example.com`. The incorrect origin update happens here. For the old `document` to be unloaded (which is not really unloaded but left being dangled), Chrome should not have updated its origin. However, it incorrectly updated to the new origin, `http://example.com`.
- ④ When the new `document` finishes loading, it fires the `load` event and invokes the event handler. The handler uses the `Function` constructor of the old `document` to execute the script in the context of the old `document`. The script changes the location of the `iframe` to a JavaScript URL, which is equivalent to executing the script in the `iframe`'s `document`.

Normally, this should have been blocked per the same-origin policy—i.e., the context of the old `document`, whose origin is `http://attacker.com`, cannot access the new `iframe`'s `document`, whose origin is `http://example.com`. However, due to the incorrect origin update to the old `document`, the payload (i.e., `'alert(document.cookie)'`) was executed in the target's context, thereby reading the target's cookies.

4 Design

Now we present the design of the FuzzOrigin. First, we introduce the overall design and workflow of FuzzOrigin (§4.1). Next, we present the origin sanitizer, which is designed to detect UXSS vulnerabilities (§4.2). As noted before, UXSS detection is challenging because it is a semantic vulnerability, which requires interactive or navigation semantics among cross-origin pages. The origin sanitizer addresses such a challenge by keeping track of origin semantics as the DOM tree is updated. Then it checks if the origin semantics are correctly updated when executing the script. Lastly, we describe how FuzzOrigin generates HTML/JavaScript inputs to prioritize origin-update operations, thereby effectively finding UXSS vulnerabilities (§4.3).

4.1 Overview

The overall design and workflow of FuzzOrigin are illustrated in Figure 4. FuzzOrigin generates random HTML files (embedding JavaScript), where the generation algorithm is aware of the HTML/JavaScript grammar [2, 66] (marked ①). When generating, FuzzOrigin performs the following two unique tasks: (i) FuzzOrigin instruments additional check code to detect UXSS, which we call origin sanitizer (§4.2); and (ii) FuzzOrigin prioritizes origin-update operations so as to increase the chance to trigger UXSS (§4.3). Then generated HTML files are deployed to a set of preconfigured servers, in which each HTML is intended to navigate to

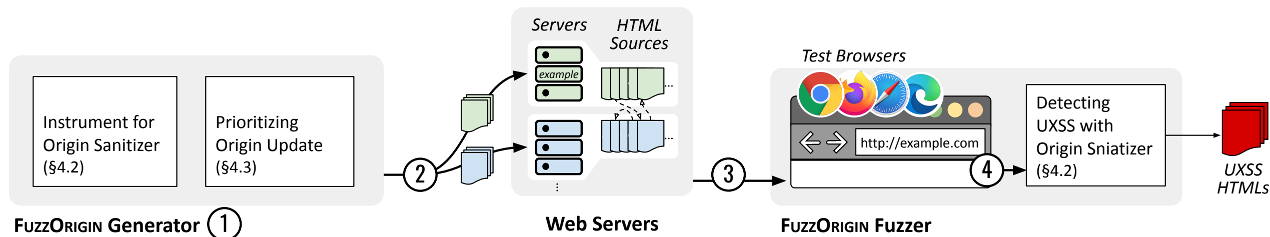


Figure 4: Overall design and workflow of FUZZORIGIN.

other HTMLs (②). Next, FUZZORIGIN runs a web browser with one of the randomly picked server's URL (③). As the browser loads the HTML, FUZZORIGIN's origin sanitizer (which was instrumented before) constantly checks if UXSS has occurred (④). If the UXSS is detected by the origin sanitizer, then FUZZORIGIN reports such an HTML case as a UXSS vulnerability.

4.2 Detecting UXSS with Origin Sanitizer

UXSS is a semantic vulnerability, which occurs if the browser incorrectly updates the origin in `document` when updating the DOM tree in response to various events. In order to detect the UXSS vulnerability, we first clearly define the primitive security property for UXSS, *origin violation* as follows.

Definition: Origin violation. The origin when executing a script block (which we denote as $\texttt{origin}_{\texttt{Exec}}$) should be the same as the origin initially assigned for the script block when fetching the HTML document (which we denote as $\texttt{origin}_{\texttt{Fetch}}$). We state that the script execution raises *origin violation* if $\texttt{origin}_{\texttt{Exec}}$ and $\texttt{origin}_{\texttt{Fetch}}$ are different.

Interpreting the previous UXSS vulnerabilities (described in §3.1 and §3.2) with the notion of the origin violation, $\texttt{origin}_{\texttt{Fetch}}$ is represented with the background color of `script` blocks (Figure 2c and Figure 3c). $\texttt{origin}_{\texttt{Exec}}$ follows the origin of the `document`, the first parent `document` of the to-be-executed `script` block. As such, both previous UXSS vulnerabilities raise the origin violation as $\texttt{origin}_{\texttt{Fetch}}$ (i.e., `http://www.attacker.com`) and $\texttt{origin}_{\texttt{Exec}}$ (i.e., `http://example.com`) are different, allowing attacker's script to be executed on behalf of the target's origin.

Given this definition, the requirements for UXSS detection would reduce down to the following three tasks: 1) how to keep track of $\texttt{origin}_{\texttt{Fetch}}$ until the point of script execution; 2) how to retrieve $\texttt{origin}_{\texttt{Exec}}$ when executing the script; and 3) how to faithfully check the differences between $\texttt{origin}_{\texttt{Fetch}}$ and $\texttt{origin}_{\texttt{Exec}}$ at all script execution points. In the following, we describe how FUZZORIGIN handles each task in turn.

Tracking $\texttt{origin}_{\texttt{Fetch}}$ with Static Tagging. When the browser fetches the HTML document, $\texttt{origin}_{\texttt{Fetch}}$ is determined and stored as a property in `document`. The goal here is then to keep track of $\texttt{origin}_{\texttt{Fetch}}$ until the point of script execution. Note that the tracking of $\texttt{origin}_{\texttt{Fetch}}$ is quite a challenging task—e.g., the browsers still suffer from UXSS vulnerabilities due to this challenge.

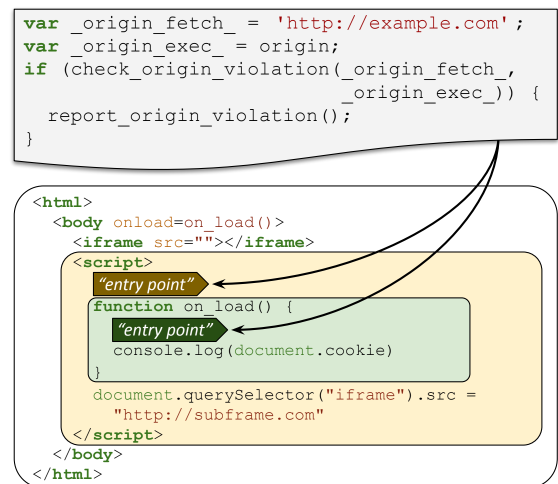


Figure 5: An example of Origin Sanitizer's instrumentation, inserting the check code to global and functional entry points.

The core idea to trace $\texttt{origin}_{\texttt{Fetch}}$ is to statically tag $\texttt{origin}_{\texttt{Fetch}}$ within the script, which is not updated over the entire life-cycle of rendering. In particular, FUZZORIGIN first determines $\texttt{origin}_{\texttt{Fetch}}$ for each HTML document to be served to the browser. Because FUZZORIGIN controls the server URL along with its port to serve the HTML document, FUZZORIGIN can always determine $\texttt{origin}_{\texttt{Fetch}}$ for each generated HTML. Then FUZZORIGIN tags $\texttt{origin}_{\texttt{Fetch}}$ inside scripts, which declares a new variable having the $\texttt{origin}_{\texttt{Fetch}}$ string as value. This declaration is performed for every point that the origin should be checked, which will be explained later in this section.

```
1 var _origin_fetch_ = 'http://example.com';
```

Therefore, although FUZZORIGIN does not explicitly trace the $\texttt{origin}_{\texttt{Fetch}}$, this static origin tagging method allows FUZZORIGIN to obtain a correct $\texttt{origin}_{\texttt{Fetch}}$ depending on the JavaScript execution context.

Retrieving $\texttt{origin}_{\texttt{Exec}}$. It is quite simple to retrieve $\texttt{origin}_{\texttt{Exec}}$, which can be done by reading the `origin` property in JavaScript. This is because the browser implements the interface of the `origin` property so as to allow the script

code to check its origin of the execution at runtime. Note that this origin property is supported by most of modern web browsers (including Chrome, Firefox, Edge, and Safari) as it is dictated in the HTML standard [41].

Checking Origin Violation. Given the capability of obtaining $\texttt{origin}_{\texttt{Fetch}}$ and $\texttt{origin}_{\texttt{Exec}}$, now we describe how FuzzOrigin checks the origin violation. FuzzOrigin checks the origin violation for all possible entry points of script code execution. As described in §2.2, the browser renderer executes the script in response to browser events, so there can be multiple execution entry points within <script>. These execution entry points include (i) a global entry point (which is executed right after parsing the script tag), (ii) functional entry points (i.e., the renderer hands over the execution to a certain function), or (iii) dynamic entry points evaluating string code (i.e., eval and Function in Figure 3b).

For all possible entry points, FuzzOrigin instruments the code to check the violation (illustrated in Figure 5). FuzzOrigin first obtains $\texttt{origin}_{\texttt{Fetch}}$, which is statically tagged before. Then it obtains $\texttt{origin}_{\texttt{Exec}}$ through accessing the origin property. Then check_origin_violation() performs the origin violation check. It returns true if the $\texttt{origin}_{\texttt{Fetch}}$ and $\texttt{origin}_{\texttt{Exec}}$ are different, implicating that FuzzOrigin detected UXSS. It returns false otherwise. One exception is matching the null origin (i.e., an opaque origin). Specifically, if $\texttt{origin}_{\texttt{Exec}}$ is null, FuzzOrigin returns false even if two origins are different. This is because the null origin can be created during navigation (e.g., loading a data URL) and it has no capability to access pages other than pages with the null origin.

If the violation is detected, FuzzOrigin reports the violation. The report includes the point where the origin violation is raised as well as the stack trace of the violation. FuzzOrigin also includes all the generated HTML files and server setups, which allows users of FuzzOrigin to reproduce a discovered UXSS vulnerability if needed.

It is worth noting that while the instrumentation for global and functional entry points seem straight-forward, one for dynamic entry points may seem unclear. However, since FuzzOrigin generates HTML with its complete abstract syntax tree, FuzzOrigin can always locate the dynamic entry points. Once located, FuzzOrigin prepends the serialized string of the origin-violation checking code right before the original string to be evaluated.

4.3 Prioritizing Origin-Update Operations

We observed that UXSS vulnerabilities occur due to incorrect origin updates in the DOM tree (§3). Based on this observation, FuzzOrigin attempts to prioritize origin-update relevant operations when generating HTML inputs. The idea behind prioritizing origin-update operations is in performing more frequent navigation operations while handling associated events. Revisiting previous UXSS vulnerabilities in the case study (§3), origin updates take place while performing cross-origin navigations such as loading, redirecting, or reloading. Then the origin violation occurs when executing the script, which is triggered in response to events dispatched by the navigation operations.

To this end, the HTML generation of FuzzOrigin is designed to meet the following two goals: i) raising cross-origin navigation; and ii) chained navigation with event handlers. Next, we describe how FuzzOrigin meets each goal in turn.

Raising Cross-Origin Navigation. A browser updates the origin as it navigates to a different, cross-origin web page. Thus, in order to test as many navigation actions as possible in the browser, the HTML generation of FuzzOrigin considers the following two things: 1) use various navigation APIs; and 2) specify cross-origin navigation targets.

First, FuzzOrigin generates HTML in consideration of a complete list of navigation-relevant APIs (i.e., $\texttt{API}_{\texttt{nav}}$). Table 1 shows the list of navigation APIs, which can be categorized into the APIs with HTML attributes and ones without HTML attributes. Navigation APIs using the HTML attributes (i.e., href of the <a> tag, action of the form tag, and src of the iframe tag) specify the target URL to be navigated once triggered. As the navigation trigger for HTML attributes may vary (i.e., <href> requires a click action, and form requires a submit action), FuzzOrigin accordingly generates associated action-triggers with the JavaScript code. It is worth noting that these HTML attributes can be statically generated (i.e., embedding HTML tags) or dynamically added (i.e., inserting an element with the script execution), and thus FuzzOrigin randomly alternates static and dynamic generation cases.

Navigation APIs without HTML attributes can be invoked through JavaScript, which includes history (i.e., moving forward, backward, or replacing the history state), location (i.e., replacing or loading the current location), or opening the window.

Second, when invoking navigation APIs, FuzzOrigin specifies various cross-origin navigation targets. It is worth noting that the following three navigation APIs do not take the target URL to be navigated, as it does not need to. For instance, history.forward and history.backward navigate to forward and backward, respectively, and location.reload() reloads the current page. If the navigation APIs take the target URL, FuzzOrigin randomly selects a URL from a prepared pool of URLs. Such a URL pool is initialized with a preconfigured set of multiple web servers, where each server again includes multiple web pages. As a result, this pool setup allows FuzzOrigin to test cross-origin navigation of browsers.

Chained Navigation with Event Handlers. Once navigation APIs are invoked, the navigation actions are performed by the browser, which in turn fires navigation events. In particular, the browser fires a specific set of events associated with each navigation API (listed in Table 1).

Navigation APIs ($\texttt{API}_{nav}$)	Type	Generation	Target URL	Triggering Action	Dispatched Events ($\texttt{Event}_{nav}$)
`a.href=URL`	Attribute	HTML/JavaScript	O	Click	beforeunload, unload, DOMContentLoaded, load
`form.action=URL`	Attribute	HTML/JavaScript	O	Submit	beforeunload, unload, DOMContentLoaded, load
`iframe.src=URL`	Attribute	HTML/JavaScript	O	-	DOMContentLoaded, load
`history.forward()`	Method	JavaScript	X	-	beforeunload, unload, DOMContentLoaded, load
`history.backward()`	Method	JavaScript	X	-	beforeunload, unload, DOMContentLoaded, load
`history.replaceState(state, title, URL)`	Method	JavaScript	O	-	beforeunload, unload, DOMContentLoaded, load
`location.replace(URL)`	Method	JavaScript	O	-	beforeunload, unload, DOMContentLoaded, load
`location.reload()`	Method	JavaScript	X	-	beforeunload, unload, DOMContentLoaded, load
`window.open(URL)`	Method	JavaScript	O	-	beforeunload, unload, DOMContentLoaded, load

Table 1: A list of navigation APIs (i.e., $\texttt{API}_{nav}$). The column 'Generation' represents if $\texttt{API}_{nav}$ can be used as an HTML tag or invoked using JavaScript. The column on 'Target URL' shows if the $\texttt{API}_{nav}$ takes the target URL parameter or not. The column on 'Triggering Action' denotes an extra action required to trigger a navigation behavior of $\texttt{API}_{nav}$. The column on 'Firing Event' shows a list of events fired by the corresponding $\texttt{API}_{nav}$. Note the beforeunload and unload event of `window.open(URL)` is dispatched when an existing `window` is reused.

As such, FuzzOrigin randomly registers multiple event handlers associated with navigation APIs. The events `beforeunload` and `unload` are dispatched before and after unloading the page. The event `load` and `DOMContentLoaded` are dispatched when loading is completed. FuzzOrigin defines these four events as $\texttt{Event}_{nav}$ and uses those to handle navigation events.

Within each event handler, FuzzOrigin then randomly invokes another navigation APIs so as to chain the navigation behaviors. This chaining makes the browser keep navigating through cross-origin web pages under various circumstances, further extending the testing coverage towards browser's origin update logic.

It is worth noting that always invoking $\texttt{API}_{nav}$ and registering $\texttt{Event}_{nav}$ would not lead to UXSS conditions. This is because the browser may not perform meaningful operations only with these APIs and events. Therefore, FuzzOrigin provides `WEIGHTED_RAND` (Algorithm 2) as a configuration parameter, balancing the API invocation (i.e., between $\texttt{API}_{nav}$ and non-$\texttt{API}_{nav}$) as well as the event registration (i.e., between $\texttt{Event}_{nav}$ and non-$\texttt{Event}_{nav}$). Specifically, if `WEIGHTED_RAND` is zero, all possible APIs (including $\texttt{API}_{nav}$) and all possible events (including $\texttt{Event}_{nav}$) will be invoked and registered, respectively. In this configuration setup, FuzzOrigin does not prioritize the chained-navigation. The number of all possible APIs can vary depending on the number of DOM instances/parameters/methods, but it is mostly over 600 APIs. The number of all possible events is 89. If `WEIGHTED_RAND` is one, on the contrary, only $\texttt{API}_{nav}$ and $\texttt{Event}_{nav}$ will be invoked and registered, respectively. In this configuration, FuzzOrigin maximizes the prioritization using nine $\texttt{API}_{nav}$ and four $\texttt{Event}_{nav}$.

To generate HTML, FuzzOrigin focuses on `new function`, `new eventhandler` and web APIs. We present the detailed algorithm of HTML generation in Appendix A for reference.

5 Implementation

We implemented FuzzOrigin, which is able to test most modern web browsers, Chrome, Firefox, Safari, and Edge. In terms of the implementation complexity, FuzzOrigin is implemented in about 9k lines of Python code. (3.5k LoCs are HTML and Javascript generation, and 2.5k LoCs are for browser testing frameworks.) FuzzOrigin is open-source and available at `https://github.com/compsec-snu/fuzzorigin`.

Origin Sanitizer and HTML Generator. To generate HTML and CSS, we used Domato [12] which is a state-of-the-art generation-based DOM fuzzer. For JavaScript, we implemented our own JavaScript generator for FuzzOrigin similar to Fuzzil [14]. In order to generate syntactically and semantically correct JavaScript, we defined JavaScript grammar (e.g., for, if, function statement) and DOM API (e.g., document.createElement()) as Python classes. However, it is an open and challenging problem to generate HTML/JavaScript covering entire HTML/JavaScript grammars, and we will discuss it in §7. The origin sanitizer of FuzzOrigin is also implemented in Python within the JavaScript generator.

Browser Testing Framework. In order to perform automated browser testing, we used Python selenium library and WebDriver. By using the WebDriver, FuzzOrigin can check the violation report of the origin sanitizer without browser modification. The testing servers are implemented using the Python flask library, and are created and managed with Docker to handle a large number of servers.

6 Evaluation

This section attempts to evaluate FuzzOrigin with the following focuses:

- Performance of the origin sanitizer to detect UXSS, in terms of detection accuracy overhead (§6.1)
- Effectiveness of chained-navigation with event handling to prioritize origin update operations (§6.2)
- New vulnerabilities discovered by FuzzOrigin. (§6.3)

Experimental Setup. We ran FUZZORIGIN on Intel Xeon Silver 4214R (24 cores) with 512 GB RAM. We prepared five web servers (i.e., five origins) and 10 HTML files for each server, totaling 50 HTMLs for a fuzzing iteration.

In order to comprehensively evaluate FUZZORIGIN's effectiveness, we compared FUZZORIGIN with Domato [12], and Freedom [67], which are state-of-the-art DOM fuzzers. Since Domato and Freedom cannot detect UXSS and thus cannot keep track of origin changes, we incorporated FUZZORIGIN's origin sanitizer to those for fair comparison.

6.1 Performance of Origin Sanitizer

UXSS Detection Accuracy. FUZZORIGIN uses tagged origin (i.e., $\texttt{origin}_{\texttt{Fetch}}$) as an oracle to compare with the $\texttt{origin}_{\texttt{Exec}}$. Thus, there are no false positive cases (i.e., origin sanitizer detects an origin violation but it was not a UXSS vulnerability), unless the tagged origin is incorrectly determined. As the $\texttt{origin}_{\texttt{Fetch}}$ can always be determined when generating the HTML file, we argue that FUZZORIGIN's origin sanitizer is free from false positives. This argument can be indirectly supported by our evaluation experiences over six months of running FUZZORIGIN, because we were not able to find any origin violation report other than the four cases we reported and confirmed. All the reported four vulnerabilities were confirmed and we could not find any false-positive cases.

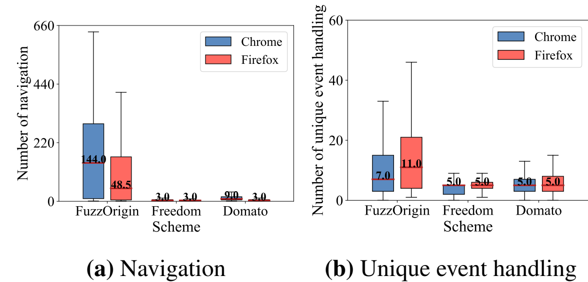
Runtime Detection Overhead. In order to analyze the performance overhead of origin sanitizer, we measured the execution time and the invocation number of the origin comparisons per each fuzzing iteration. The average execution time per one origin comparison was 0.0098 ms, and a single fuzzing iteration has invoked 219.58 origin comparisons on average. Consequently, origin sanitizer uses 2.16 ms for the origin comparisons in each fuzzing iteration, which is 0.11% of the total execution time (i.e., 2.00 s). Considering that the network latency for fetching the HTML file is usually around 100 ms, the 2.16 ms overhead per a fuzzing iteration is reasonable.

6.2 Effectiveness of Chained-Navigation

FUZZORIGIN created HTML in the direction of using navigation and event handler many times to find UXSS vulnerabilities. In this respect, we analyze whether the navigation and event handler calls were frequently triggered as intended. And we analyze $\texttt{origin}_{\texttt{Exec}}$ updating count in the browser to find which of the navigation or event is more important in influencing the origin-update.

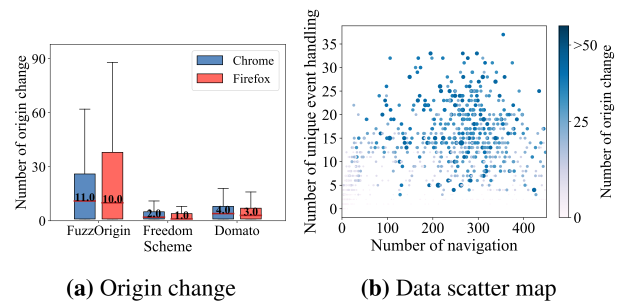
6.2.1 Navigation and Event Handler

Navigation. We measured the number of navigations using how many requests were received by the servers. The empirical distribution of navigation counts is presented as a box plot in Figure 6a for FUZZORIGIN, Freedom, and Domato. FUZZORIGIN achieves the highest median in comparison to Freedom and Domato.



(a) Navigation **(b)** Unique event handling

Figure 6: The number of navigation completion and unique event handling for a single fuzzing iteration by each scheme. In both cases, FUZZORIGIN achieved the highest number.



(a) Origin change **(b)** Data scatter map

Figure 7: The number of origin updates per each fuzzing run, and the scatter map showing the impacts of navigation and event handling behaviors to origin changes.

Event Handler. We measured the number of event handlers directly. Figure 6b describes the number of unique calls to the event handler. Compared to Freedom, FUZZORIGIN made 1.2 times more unique event handler calls in Chrome and 2.2 times more in Firefox. Freedom and Domino use a fixed number of event handlers and execute that only once. On the other hand, FUZZORIGIN not only executes all event handlers that are called, but also has a higher number of unique event handlers than the other two schemes.

6.2.2 Origin-Update

Execution Origin-Update. We measured origin-update by $\texttt{origin}_{\texttt{Exec}}$ to evaluate FUZZORIGIN. Figure 7a is the result of $\texttt{origin}_{\texttt{Exec}}$ changing. FUZZORIGIN recorded the largest number of origin-update. The results are almost similar to those of navigation, but the difference is significantly reduced. However, this value is still the largest value and shows that the $\texttt{origin}_{\texttt{Exec}}$ is sufficiently changed through navigation and event handler as originally intended by FUZZORIGIN.

Correlation with Navigation & Event Handler. Figure 7b shows the overall data patterns. The X-axis represents the number of triggered navigations and the Y-axis represents the

Variable	Firefox	Chrome
(Intercept)	-1.800* (0.479)	-0.502 (0.394)
Navigation	1.338* (0.029)	1.329* (0.036)
Event Handler	0.025* (0.002)	0.020* (0.002)
R-squared	0.733	0.671

Table 2: Result of Poisson regression. R^2 represents that overall regression was statistically significant. Navigation and event handler are significant predictors in both Firefox and Chrome. (Standard errors are reported in parentheses. * indicates significance at the 99% level.)

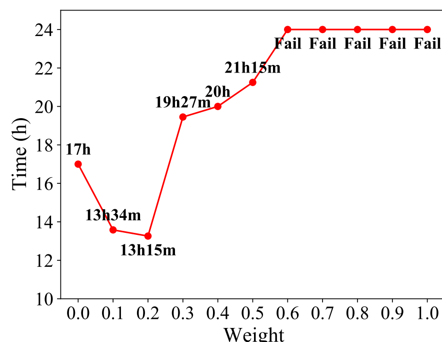


Figure 8: An elapsed time in average to detect seven previously known CVEs while varying `WEIGHTED_RAND`. When `WEIGHTED_RAND` is higher than 0.6, no CVEs were found within 24 hours of running.

number of invoked unique event handlers. And the number of origin-updates is shown as color. It can be seen that the color of the point gets darker as the distance from the x-axis and y-axis increases.

We analyze correlation more deeply with Poisson regression to which of them is more important between navigation and event handler. Table 2 is the result of the fitted regression model. R^2 was 0.733 and 0.671 which indicates the overall regression was statistically significant. The event handler is a statistically significant predictor (with coefficients 0.025 and 0.020) in both Firefox and Chrome Navigation was also reported to be significant (with coefficients 1.338 and 1.329) in both browsers.

In summary, to increase origin-update, both navigation and event handlers were analyzed to be significant. The design of FuzzOrigin that generates HTML by combining navigation and event handler is an effective strategy to make many origin changes.

6.2.3 Detecting Previously Known UXSS

In this experiment, we check if FuzzOrigin is able to find seven previously known UXSS vulnerabilities to show the effectiveness of chained-navigation. These include CVE-2016-1667, CVE-2016-1697, CVE-2016-1711, CVE-2016-5204, CVE-2016-5207, CVE-2016-5208, and CVE-2017-5008, and all of these CVEs can be reproduced in a single Chrome binary (i.e., Chrome 52.0.2715). Since it takes a very long time for FuzzOrigin to randomly generate the exploitation code, we conducted this experiment by turning FuzzOrigin into a mutation-based fuzzer with following two rules: i) an initial HTML template per known CVE is provided to FuzzOrigin, where the template is the known PoC HTML where its JavaScript APIs have been wiped out, and ii) given the template, FuzzOrigin keeps replacing wiped out entries with APIs according to `WEIGHTED_RAND`.

Figure 8 shows an elapsed time in average to find seven UXSS cases while varying `weight` value. FuzzOrigin took the shorted time when `WEIGHTED_RAND` is 0.2 (i.e., 20%). Compared to the case that origin-update prioritization is not used at all and all the APIs were randomly selected (i.e., `WEIGHTED_RAND` is zero), FuzzOrigin was 3 hours and 45 minutes faster when `WEIGHTED_RAND` is 0.2. On the contrary, compared to the case that FuzzOrigin always uses either $\texttt{API}_{\texttt{nav}}$ and $\texttt{Event}_{\texttt{nav}}$ (i.e., `WEIGHTED_RAND` is 1), FuzzOrigin failed to find any CVEs for the given 24 hours. This is because if FuzzOrigin generates too many $\texttt{API}_{\texttt{nav}}$, navigation operations are performed even before any meaningful API sequences are constructed. To summarize, these results suggest that chained-navigation indeed helps FuzzOrigin to find UXSS vulnerabilities if a right balance between origin-update APIs and normal APIs were given. While finding an optimal balance would also be important for FuzzOrigin, we leave this as a future work.

6.3 New Vulnerabilities Discovered by FuzzOrigin

We ran FuzzOrigin about six months to test and find UXSS vulnerabilities. During the evaluation, FuzzOrigin found four new vulnerabilities in total (Figure 9). FuzzOrigin found two vulnerabilities that could run UXSS by changing the port in Chrome (i.e., Issue #1280083)[4] [5] and Firefox (i.e., Issue #1741327). Both vulnerabilities have different PoCs, but we suspect the root causes are in `document.domain`. Moreover, FuzzOrigin found two vulnerabilities in Firefox, one of which is classified as a high-impact security vulnerability (i.e., CVE-2021-43536). Firefox has patched it and issued a security advisory. Another one (i.e., Issue #1727480) is not patched yet due to low reproducibility. We note that this vulnerability was triggered due to FuzzOrigin's chained-navigation fuzzing.

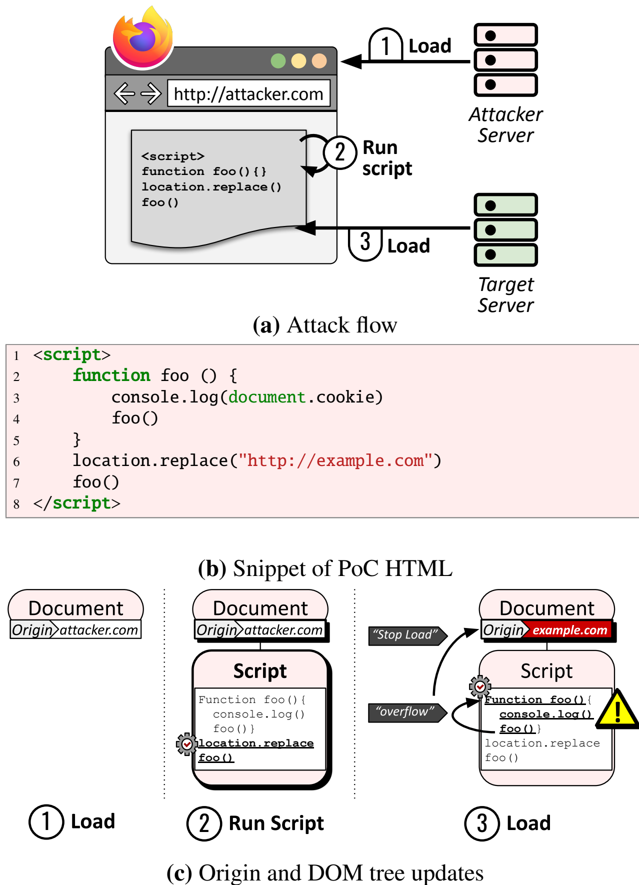
Case Study: CVE-2021-43536. This vulnerability occurs when the document loader fails to initialize due to an error (e.g., a possible stack overflow in the case of our PoC).

[4] Chromium-based browsers such as Edge are also vulnerable to this vulnerability.

[5] After being confirmed, a Chromium developer commented that the reported Issue #1280083 is not UXSS because it requires a specific security relaxation. However, we think this still is UXSS as such relaxation is quite common in practice–Google reported that 13% of web sites have such relaxation [52].

Browser	Version	Bug ID	Description	Severity	Status
Chrome	96.0.4664	Issue #1280083	document.domain used in parent and child causes the origin (port) change.	Low	Confirmed
Firefox	94.0b2	Issue #1741327	document.domain used in parent and child, causes script execution even if src of child window to the parent's origin.	Serious	Confirmed
	94.0b9	Issue #1727480	History manipulation causes navigation to other pages on nsDocShell.	Serious	Confirmed
		CVE-2021-43536	Under certain circumstances, asynchronous functions could have caused navigation to fail but expose the target URL.	High	Pached

Figure 9: A list of vulnerabilities found by FUZZORIGIN.



(a) Attack flow

(b) Snippet of PoC HTML

(c) Origin and DOM tree updates

Figure 10: New vulnerability: CVE-2021-43536.

Figure 10b is the snippet of a PoC code[6]. On the PoC code, foo function (lines 2-5) calls itself (line 4). The location is changed to `http://example.com` (line 6), and then foo is called (line 7) while the page is loading. The function foo is called recursively, causing the stack to fill up. If this is just before the `http://example.com`, the document loader fails to initialize and only the origin is updated to `http://example.com`. Then, because the origin has changed, the cookie of `http://example.com` is displayed (line 3).

This vulnerability could be only found with FUZZORIGIN, as it dynamically creates and calls the function, whereas most DOM fuzzers rarely generate functions that is used as event handlers.

[6] The actual PoC code is much more complicated, but we simplified it to clearly show the root cause and attack flow.

Case Study: #1727480. We will briefly explain issue #1727480 in the abstract since it has been confirmed but has not been fixed yet. `nsDocShell` [35] is responsible for loading and viewing of a document in Firefox. Issue #1727480 is caused by origin confusion in `nsDocShell` when navigating via history.back() and history.forward() inside iframe. Figure 11b is the snippet of the PoC code[6]. `http://attacker.com` has two iframes embedding `http://example.com` (line 3) and creates a new iframe with an `unload` event handler (lines 9-11). The function payload (lines 5-8) will be triggered when the created iframe is unloaded. `http://example.com` has two $\texttt{API}_{\texttt{nav}}$: history.back(), and history.forward(). By some unknown cause, this caused the parent window to navigate to `http://example.com`. However, the actual navigation did not take place, leaving the onload event handler to be executed in the context of `http://example.com` origin.

Case Study: #1280083 and #1741327. These two were detected in Firefox and Chrome, respectively, but the pattern of PoC is similar. According to the MDN, `document.domain` is deprecated [37]. In particular, MDN warns that changing the value of document.domain deletes port information, which is dangerous from the security point of view. Since port information disappears, resources can be accessed from cross-origin with a different port. If certain conditions are met, XSS can also be performed with cross-origin by changing the port. For instance, in a shared hosting and cloud setup, two different websites may share the same IP address but use different ports.

7 Discussion and Limitation

UXSS Detection without the Origin Sanitizer. There can be alternatives of the origin sanitizer to detect UXSS, but these have its own limitations compared to the origin sanitizer. One approach is to generate attacker's HTML and check whether the JavaScript is executed under the origin of the victim. However, it cannot find a vulnerability that requires to generate HTMLs of both attacker and victim, such as #1280083 and #1741327 found by FUZZORIGIN. Another approach would be to have an attacker steal victim's resources (e.g., cookie). However, this approach would have following two issues. First, there would be false positives if the script is dynamically evaluated through `eval()`. In this case, one cannot determine where the JavaScript is fetched from and where it is executed. Second, the integrity of the

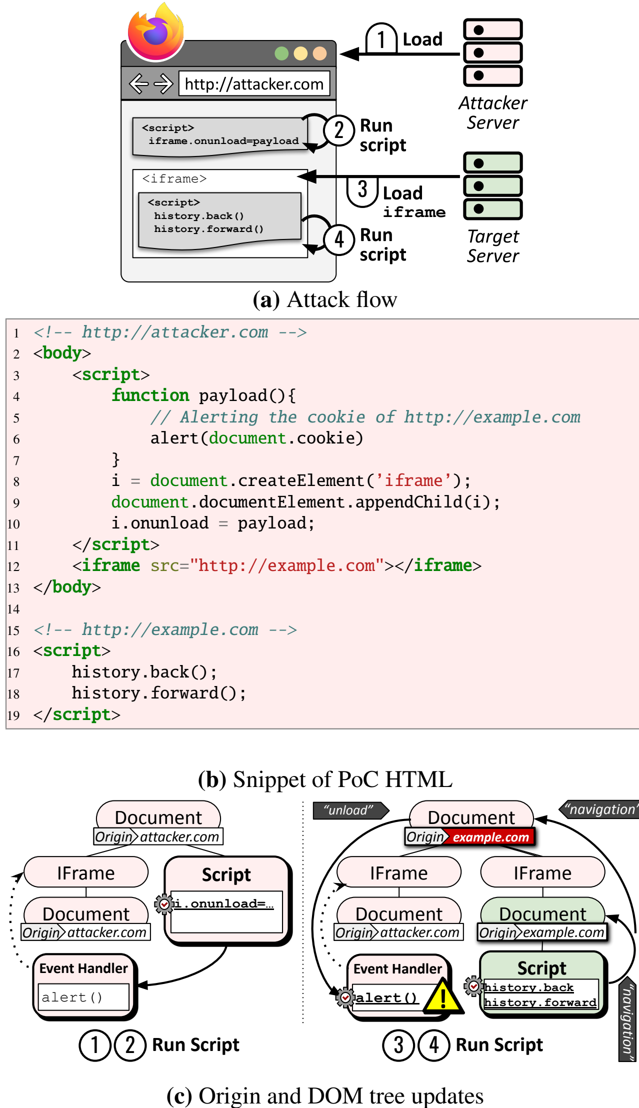


Figure 11: New vulnerability: #1727480.

victim's token value has to be ensured, but as FUZZORIGIN dynamically generates the JavaScript, such integrity can be violated at runtime. One may be able to fix this issue by restricting the JavaScript random generation process, which will need a further study.

UXSS Mitigation with the Origin Sanitizer. While FUZZORIGIN leveraged the origin sanitizer for UXSS fuzzing, we think it has potential to be used to prevent UXSS attacks in web browsers. Most browsers manage origins according to the HTML specification [65], but it is extremely challenging to implement all of those correctly. The origin related policies are already complex, involving various corner-cases, and thus modern browsers still have critical UXSS vulnerabilities. Employing FUZZORIGIN's origin sanitizer would help to address this issue, but that would still entail additional research challenges which require further studies—e.g., tracking all $\texttt{origin}_{\texttt{Fetch}}$ is difficult due to the JavaScript dynamic interpretation (e.g., `eval`).

Non-deterministic Behaviors of the Browser. When analyzing the vulnerabilities FUZZORIGIN found, we observed that the non-deterministic behavior of the browser is related to UXSS. This is mainly due to the non-deterministic latency in loading each page. Specifically, since FUZZORIGIN heavily triggers navigation operations as well as associated events, the order of page loading often becomes non-deterministic as well. This rendered the vulnerability reproduction difficult, so it is challenging to perform the detailed vulnerability analysis. For instance, the vulnerability #1727480 had quite a low reproducibility due to this issue. We think this is an interesting finding that the order of events or timings may impact the overall behaviors of browsers, which would be worth the further study, possibly from the fuzzing research perspective.

Limitation of HTML Generation. As the HTML/-JavaScript syntax is complex, the current implementation of FUZZORIGIN to generate HTML is limited. We observed that there are certain types of JavaScript code patterns that the current FUZZORIGIN cannot generate–such as the code pattern using the API of the JavaScript engine. Currently FuzzOrigin supports `eval`, `setTimeout`, `new function`, `new eventhandler`, and `XMLHTTPRequest` in JavaScript. In addition, all the tags and attributes of HTML are supported, and cross-origin loading is possible if they have an src attribute. However, FUZZORIGIN cannot certain code patterns using the API of the JavaScript engine, such as `prototype`, `promise`, and `arrow function`. FUZZORIGIN cannot cover other resources (e.g., browser extensions and bookmark features), which can trigger UXSS vulnerabilities. Covering all complex HTML/JavaScript code patterns and resources would definitely help FUZZORIGIN's UXSS detection capability, and we leave this task as a future work.

8 Related work

Universal Cross-site Scripting (UXSS). There have been few studies related to the UXSS vulnerabilities. Barth et al. [4] identified cross-origin JavaScript capability leaks and proposed an algorithm which monitors the points-to relation of the JavaScript heap for detecting such vulnerabilities. However, this approach cannot be applied to detect the UXSS vulnerabilities caused by incorrect origin checks—the major reason of UXSS vulnerabilities. Recently, Moroz et al. [31] analyzed the bugs in the Chrome browser which lead to the UXSS vulnerabilities. Compared to these, FUZZORIGIN is the first, automatic framework to find the UXSS vulnerabilities.

While not directly focusing on UXSS, there were several previous works discussing security issues highly related to UXSS. These include the same-origin policy (SOP) [54],

cookies [5, 9–11, 57, 70], and cross-origin resource sharing (CORS) [6, 29]. Schewenk et al. [54] developed a comprehensive testing framework to test SOP for DOM tree accesses. Franken et al. [11] evaluated the access policies for third-party cookies to prevent cross-site attacks or third-party tracking. Drakonakis et al. [9] conducted the study of cookie-based account hijacking in the wild. However, none of these techniques was applicable to find the UXSS vulnerabilities.

Browser Fuzzing. FuzzOrigin performs the HTML fuzzing to find UXSS, a semantic vulnerability in browsers. However, most previous works performing similar HTML fuzzing and JavaScript engine fuzzing are designed to find memory bugs. As such, these focused on testing DOM construction and modification routines of browsers, which are well-known memory bug sources.

Most existing DOM fuzzers [12, 32–34, 69] have taken the generation-based fuzzing approach. Cross-fuzz [69] generates an extremely long-winding sequence of DOM operations and creates circular references to stress the browser's memory management. Domato [12] is a state-of-the-art fuzzer which generates grammatically correct HTML documents based on predefined grammar files to test Chrome browser. Dharma [33] and Avalanche [32] generated inputs based on the context-free grammars provided by Mozilla. Recently, Freedom [67] introduced an approach to efficiently generate HTML by relying on a context-aware intermediate representation. Freedom [67] stated that the coverage feedback is not helpful to find more bugs.

Previous works for JavaScript Engine fuzzing [14, 17, 24, 47] have focused on generating semantically correct JavaScript. Montage [24] and DIE [47] leveraged abstract syntax trees (ASTs) for mutation. CodeAlchemist [17] proposed semantic-aware assembly, and Fuzzil [14] designed intermediate representation (IR) to build syntactically and semantically correct test cases.

Fuzzing for Semantic Vulnerabilities. There were previous works which fuzzing techniques to find semantic bugs. To find the semantic bugs, many studies [7, 15, 19, 21, 50, 63, 68] leveraged differential testing techniques. Nezha [50] proposed an efficient input-format-agnostic differential testing framework to trigger semantic bugs. TCP-Fuzz [71] used differential testing to detect memory and semantic bugs in TCP stacks.

In addition to fuzzing traditional software, several frameworks that conduct fuzz testing on new targets have been introduced. Deepxplore [48] is the whitebox framework to systematically test real-world DL systems. DiFuzzRTL [18] detects semantic bugs in CPU by comparing the execution results of ISA and RTL simulation. Along the line of the previous semantic fuzzing research, FuzzOrigin introduces a new type of semantic fuzzing technique, focusing on UXSS vulnerabilities.

9 Conclusion

Universal cross-site scripting (UXSS) is a critical vulnerability in web browsers, allowing attackers to execute a malicious script on behalf of pages that should not be accessible to attackers. This paper presented FuzzOrigin, the first UXSS fuzzing framework. It proposes a new UXSS detector, the origin sanitizer, as well as a new UXSS-focused HTML generation method. During the evaluation, FuzzOrigin identified four new UXSS vulnerabilities, demonstrating its effectiveness in finding UXSS issues.

10 Acknowledgment

We thank anonymous reviewers and the shepherd Sooel Son, for insightful comments, which significantly helped to improve this paper. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone). This work was supported by National Research Foundation (NRF) of Korea grant funded by the Korean government MSIT (NRF-2019R1C1C1006095). The Institute of Engineering Research (IOER) and Automation and Systems Research Institute (ASRI) at Seoul National University provided research facilities for this work.

References

[1] *Webkit2*. https://trac.webkit.org/wiki/WebKit2 (visited on January 30, 2022).

[2] *ECMAScript 2022 Language Specification*, 2022. https://tc39.es/ecma262.

[3] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan. {NAVEX}: Precise and scalable exploit generation for dynamic web applications. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 377–392, 2018.

[4] A. Barth, J. Weinberger, and D. Song. Cross-origin javascript capability leaks: Detection, exploitation, and defense. In *USENIX security symposium*, pages 187–198, 2009.

[5] A. Cahn, S. Alfeld, P. Barford, and S. Muthukrishnan. An empirical study of web cookies. In *Proceedings of the 25th international conference on world wide web*, pages 891–901, 2016.

[6] J. Chen, J. Jiang, H. Duan, T. Wan, S. Chen, V. Paxson, and M. Yang. We still donâĂŹt have secure cross-domain requests: an empirical study of {CORS}. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1079–1093, 2018.

[7] Y. Chen, T. Su, and Z. Su. Deep differential testing of jvm implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1257–1268. IEEE, 2019.

[8] Chromium. *Issue 524074*, 2015. https://bugs.chromium.org/p/chromium/issues/detail?id=524074.

[9] K. Drakonakis, S. Ioannidis, and J. Polakis. The cookie hunter: Automated black-box auditing for web authentication and authorization flaws. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1953–1970, 2020.

[10] S. Englehardt, D. Reisman, C. Eubank, P. Zimmerman, J. Mayer, A. Narayanan, and E. W. Felten. Cookies that give you away: The surveillance implications of web tracking. In *Proceedings of the 24th International Conference on World Wide Web*, pages 289–299, 2015.

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

[12] I. Fratric. *Domato*. https://github.com/googleprojectzero/domato (visited on November 15, 2021).

[13] A. Gakhokidze and N. Kochar. *Introducing Site Isolation in Firefox*, 2021. https://blog.mozilla.org/security/2021/05/18/introducing-site-isolation-in-firefox/.

[14] S. Groß. Fuzzil: Coverage guided fuzzing for javascript engines. *Department of Informatics, Karlsruhe Institute of Technology*, 2018.

[15] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun. Dlfuzz: Differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 739–743, 2018.

[16] N. Guyen. *The Best Firefox Ever*, 2017. https://blog.mozilla.org/en/products/firefox/faster-better-firefox/.

[17] H. Han, D. Oh, and S. K. Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *NDSS*, 2019.

[18] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303. IEEE, 2021.

[19] B. Jabiyev, S. Sprecher, K. Onarlioglu, and E. Kirda. T-reqs: Http request smuggling with differential fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1805–1820, 2021.

[20] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *2009 IEEE 31st international conference on software engineering*, pages 199–209. IEEE, 2009.

[21] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 147–161, 2019.

[22] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[23] V. Kumar. *$20000 Facebook DOM XSS*, 2020. https://vinothkumar.me/20000-facebook-dom-xss/ (visited on January 30, 2022).

[24] S. Lee, H. Han, S. K. Cha, and S. Son. Montage: A neural network language {Model-Guided}{JavaScript} engine fuzzer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2613–2630, 2020.

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

[26] S. Lekies, B. Stock, and M. Johns. A tale of the weaknesses of current client-side xss filtering. *BlackHat USA*, 2014.

[27] M. Liu, B. Zhang, W. Chen, and X. Zhang. A survey of exploitation and detection methods of xss vulnerabilities. *IEEE Access*, 7:182004–182016, 2019.

[28] M. C. Martin and M. S. Lam. Automatic generation of xss and sql injection attacks with goal-directed model checking. In *USENIX Security symposium*, pages 31–44, 2008.

[29] G. Meiser, P. Laperdrix, and B. Stock. Careful who you trust: Studying the pitfalls of cross-origin communication. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 110–122, 2021.

[30] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia. Riding out domsday: Towards detecting and preventing dom cross-site scripting. In *2018 Network and Distributed System Security Symposium (NDSS)*, 2018.

[31] M. Moroz and S. Glazunov. Analysis of uxss exploits and mitigations in chromium. Technical report, 2019.

[32] Mozilla. *Avalanche*, . https://github.com/MozillaSecurity/avalanche (visited on January 9, 2022).

[33] Mozilla. *Dharma*, . https://github.com/MozillaSecurity/dharma (visited on November 15, 2021).

[34] Mozilla. *DOMFuzz*, . https://github.com/MozillaSecurity/domfuzz (visited on November 15, 2021).

[35] Mozilla. *Embedding*, . https://www-archive.mozilla.org/projects/embedding/webbrowser.html/.

[36] Mozilla. *Bugzilla 1353975*, 2017. https://bugzilla.mozilla.org/show_-bug.cgi?id=1353975.

[37] Mozilla. *Document.domain*, 2021. https://developer.mozilla.org/en-US/docs/Web/API/Document/domain.

[38] Mozilla. *Iframe*, 2021. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe.

[39] Mozilla. *JavaScript*, 2021. https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[40] Mozilla. *Location*, 2021. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Location.

[41] Mozilla. *Origin*, 2021. https://developer.mozilla.org/en-US/docs/Web/API/origin.

[42] Mozilla. *Same-origin policy*, 2021. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[43] Mozilla. *WebAssembly*, 2021. https://developer.mozilla.org/en-US/docs/WebAssembly.

[44] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *NDSS*, volume 20, 2009.

[45] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *In Proceeding of the Network and Distributed System Security Symposium (NDSSâĂŹ07*. Citeseer, 2007.

[46] NVD. *CVE-2020-35774*, 2020. https://nvd.nist.gov/vuln/detail/CVE-2020-35774.

[47] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1629–1642. IEEE, 2020.

[48] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

[49] R. Pelizzi and R. Sekar. Protection, usability and improvements in reflected xss filters. In *proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 5–5, 2012.

[50] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on security and privacy (SP)*, pages 615–632. IEEE, 2017.

[51] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 219–232, 2009.

[52] C. Reis, A. Moshchuk, and N. Oskov. Site isolation: Process separation for web sites within the browser. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1661–1678, 2019.

[53] F. RosÃl'n. *How I hacked Facebook and received a $3,500 USD Bug Bounty*, 2012. https://blog.detectify.com/2012/12/30/how-i-hacked-facebook-and-received-a-3500-usd-facebook-bug-bounty/ (visited on January 30, 2022).

[54] J. Schwenk, M. Niemietz, and C. Mainka. Same-origin policy: Evaluation in modern browsers. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 713–727, 2017.

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

[56] L. K. Shar, H. B. K. Tan, and L. C. Briand. Mining sql injection and cross site scripting vulnerabilities using hybrid program analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 642–651. IEEE, 2013.

[57] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the incoherencies in web browser access control policies. In *2010 IEEE Symposium on Security and Privacy*, pages 463–478. IEEE, 2010.

[58] B. Stock and M. Johns. Protecting users against xss-based password manager abuse. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 183–194, 2014.

[59] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns. Precise client-side protection against dom-based cross-site scripting. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 655–670, 2014.

[60] M. Ter Louw and V. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *2009 30th IEEE symposium on security and privacy*, pages 331–346. IEEE, 2009.

[61] M. Van Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *NDSS*. Citeseer, 2009.

[62] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, volume 2007, page 12, 2007.

[63] Z. Wang and S. Zhu. Symtcp: eluding stateful deep packet inspection with automated discrepancy discovery. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[64] WHATWG. *Browsing Context*, 2021. https://html.spec.whatwg.org/multipage/browsers.html#creating-browsing-contexts.

[65] WHATWG. *Origin*, 2021. https://html.spec.whatwg.org/multipage/origin.html.

[66] WHATWG. *The elements of html*, 2021. https://html.spec.whatwg.org/.

[67] W. Xu, S. Park, and T. Kim. Freedom: Engineering a state-of-the-art dom fuzzer. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 971–986, 2020.

[68] Y. Yang, Y. Zhou, H. Sun, Z. Su, Z. Zuo, L. Xu, and B. Xu. Hunting for bugs in code coverage tools via randomized differential testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 488–499. IEEE, 2019.

[69] M. Zalewski. *cross_fuzz*. https://lcamtuf.coredump.cx/cross_fuzz (visited on November 15, 2021).

[70] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver. Cookies lack integrity: Real-world implications. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 707–721, 2015.

[71] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu. {TCP-Fuzz}: Detecting memory and semantic bugs in {TCP} stacks with fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 489–502, 2021.

A HTML Generation

Algorithm 1: JavaScript generation algorithm

```
1  Function genereteScript()
       input  : tags (HTML tags), N (number of statement)
       output : script (list of statement)
2      script ← [];
3      functions ← ∅;
       /* Generate N statements                              */
4      for n = 1 to N do
           /* Pick a random value in [0, 1]                  */
5          r ← RAND([0,1])
6          if r ≤ PROB_FUNC then
               /* Function definition                        */
7              statement ← generateFunction()
8              functions ← functions ∪ statement.name
9          else if r ≤ PROB_EVENT then
               /* Event handler setting                      */
10             function ← RAND_PICK(functions)
11             event ← WEIGHTED_RAND(Event_nav, Event_all)
                generateFunction()
12         else if r ≤ PROB_BLOCK then
               /* JavaScript code block                      */
13             codeblock ← RAND_PICK(codeblocks)
14             statement ← generateCodeBlock(codeblock)
15         else
               /* General JavaScript operations              */
16             api ← WEIGHTED_RAND(API_nav, API_all)
17             statement ← createCode(api, tags)
18         APPEND(script, statement)
19     end
20 end
```

Algorithm 2: Weighted random algorithm

```
1  Function WEIGHTED_RAND()
       input  : set1 (priority set), set2 (entire set)
       output : item (selected item)
       /* Pick a random value in [0, 1]                      */
2      r ← RAND([0,1])
3      if r ≤ PROB_WEIGHT then
           /* Weighted rand                                  */
4          item ← RAND_PICK(set1)
5      else
           /* Normal rand                                    */
6          item ← RAND_PICK(set2)
7  end
```

FuzzOrigin randomly generates an entire HTML file which includes 1) HTML tags, and 2) JavaScript—We focus only on HTML tags and JavaScript generation, as the CSS does not affect the origin-related operations. For the HTML tags, FuzzOrigin randomly uses all the possible HTML tags while prioritizing $\texttt{API}_{\texttt{nav}}$ related tags over the others. The HTML tags initially construct the DOM tree inside the browser, but we mainly focus on the JavaScript generation since the HTML tags are statically applied and cannot incur dynamic origin-updates in the browser.

Thus, FuzzOrigin designs a JavaScript generation algorithm as illustrated in Algorithm 1. To be specific, the algorithm (i.e., *generateScript*) takes the generated HTML tags and the statement number (i.e., `N`) as an inputs, and outputs the script of JavaScript statements. The algorithm iteratively generates a statement which randomly belongs to one of the following four types: 1) function definition, 2) event handler attaching, 3) JavaScript code blocks, and 4) general JavaScript operations.

For the function definition, FuzzOrigin defines a function template and fills the function body by recursively calling `generateScriptr` with the small number `N` (i.e., *generateFunction* in line 7). Especially, FuzzOrigin prioritize $\texttt{API}_{\texttt{nav}}$ and event triggering APIs inside the function so that the invocation of the function can be chained into further navigation or event handling. Then, FuzzOrigin appends the defined function to the corpus (i.e., line 8, *functions*), which will be used to attach the event handler.

For attaching the event handler, FuzzOrigin randomly chooses a function from the corpus (i.e., *functions*), and attaches it as a handler of a random event (e.g., click or load). FuzzOrigin also prioritizes the navigation-related events (i.e., $\texttt{Event}_{\texttt{nav}}$), thus the completion of a navigation can frequently invoke other functions (Algorithm 2).

Next FuzzOrigin considers two JavaScript code blocks (i.e., `if-else` and `try-catch`). FuzzOrigin defines a template and fills the block by recursively calling generateScript with the small number `N` (i.e., line 14, *generateCodeBlock*)

Finally for the other general JavaScript operations, FuzzOrigin randomly generates Web APIs and `function-call` (i.e., line 17, *createCode*). The web APIs include DOM object creation (i.e., document.createElement) and DOM property set, DOM method call, timer function (i.e., setTimeout), and XMLHttpRequest. FuzzOrigin covers `Document`, `Element`, `Event`, `EventTarget`, `Node` and `Window` as a DOM object. FuzzOrigin does not generate anything other than aforementioned web APIs and `function-call` to focus on origin-related operations.

During the generation, we configure the statement to frequently use $\texttt{API}_{\texttt{nav}}$ and event triggering APIs (e.g., `Click`, and `Submit`), which help fulfilling FuzzOrigin's design philosophy (i.e., frequent navigation and chained event handling). The element defined from the HTML tags can also be accessed and updated here.

While all the statements are randomly generated, the probabilities for selecting navigation-related events and $\texttt{API}_{\texttt{nav}}$ (i.e., `WEIGHTED_RAND`) can be configured so that various fuzzer settings can be used.