Parse Me, Baby, One More Time: Bypassing HTML Sanitizer via Parsing Differentials

David Klein and Martin Johns
Technische Universität Braunschweig
{*david.klein,m.johns*}*@tu-braunschweig.de*

***Abstract*—Websites rely on server-side HTML sanitization to defend against the ever-present threat of cross-site scripting attacks. Parsing arbitrary pieces of markup to assess whether they contain an exploit payload is far from trivial. This complexity leads to divergences between the parsing results of the sanitizer and the user's browser. These so-called parsing differentials open the door for the unexplored category of mutation-based attacks. Here, an attacker abuses the sanitizer's incorrect HTML parser to either directly bypass it or coerce it to transform benign markup into a dangerous exploit payload.**

In this work, we study the prevalence of such parsing differentials and their security impact. To this end, we built a generator for HTML fragments that are difficult to parse and evaluated how 11 sanitizers across five programming languages deal with such inputs. We found that parsing differentials are commonplace, as each assessed sanitizer has at least several functional deficiencies leading to overzealous removal of benign input. Even worse, we were able to automatically bypass all but two of the 11 sanitizers, painting a dire picture of the state of server-side HTML sanitization.

1. Introduction

There are two frontiers to protect against cross-site scripting (XSS): on the client and on the server-side. Traditionally, client-side XSS protection has been seen as the difficult one, as the client offers no protection mechanisms, and writing custom sanitization code is notoriously error-prone [1]. Google, for example, directly acknowledges this fact in their report on Trusted Type adoption: "More than half of the DOM XSS root causes were due to bugs in HTML sanitizers" [2]. The academic community has also mainly focused on client-side XSS, from prevalence scanning [3–8] to studying employed protection mechanisms [1, 9], the body of work is extensive. Conversely, the exploration of server-side XSS remains notably underrepresented. Large-scale server-side security scanning is comparatively scarce, primarily due to ethical and legal challenges [10].

Due to modern server-side web development's heavy reliance on frameworks, one might assume robust defense mechanisms are in place. Such defenses could come in the form of automatic sanitizer placement, as suggested in past work [11–13]. However, after inspecting the documentation of 11 popular web frameworks about their XSS protections, we found this assumption to be lacking.

Instead, we propose to take a step back and ask the question: Is server-side HTML sanitization even possible without mangling benign input?

Over the last years, the security community realized that accurate HTML sanitization is only possible with detailed information on where in the website the sanitized result is inserted [14]. While this information is possibly available for client-side sanitization, as the currently proposed Sanitizer API shows, it is out of reach for server-side sanitizers. This context sensitivity influenced the design of the sanitizer API, which does not allow to perform a string-to-string transformation [15], declaring it generally unsafe to do so. On the server, this is the only type of transformation available, as ultimately, the sanitizer's output ends up in an HTTP response, which is text-based.

Server-side sanitization routines face an additional challenge. To accurately sanitize an HTML fragment, that is, only to remove the actively dangerous part, a sanitizer has to parse it in the same fashion as a browser. A cursory glance at the HTML specification suffices to highlight the complexity of writing such a parser. Even if the sanitizer implements the specification perfectly, this does not suffice either, as browsers can and do diverge from the specification. Therefore, to accurately sanitize, a sanitizer would have to parse its input exactly like the user's browser would. This requires information on the client's browser, the parsing mode, and the exact injection context to adjust the sanitizer's behavior accordingly. This is not supported by any server-side sanitizer. This problem is further aggravated by browsers accepting invalid HTML input. Instead of aborting the parsing process, they try to rewrite and correct the input, i.e., mutating it and changing the HTML structure in the process. While this behavior is partially specified, it adds another difficulty for the authors of parsing and sanitization routines: Their software would need to support the same behaviors to assess the security impact of HTML fragments correctly. Otherwise, it opens the door to mutation-based XSS vulnerabilities.

These issues raise two interesting questions: Is it feasible to write a sanitizer that is both accurate, i.e., does not mangle benign content, and secure? And how do popular open-source sanitizing libraries fare in this respect?
These are the questions we will answer in this work.

To assess the prevalence of parsing divergences, we first analyzed the HTML specification, selecting HTML tags and edge cases that might lead to interesting parsing behavior. We then present `MutaGen`, an HTML fragment generator with a special focus on fragments prone to mutations, and

evaluate such fragments on our testbed. Here, we sanitize each generated fragment with 11 different sanitizers and evaluate their outputs in all major browsers. We also record the DOM-like structure resulting from both the sanitizers as well as the browsers' parsing processes. This allows us to automatically assess if and where parsing behavior diverges and how this can lead to sanitizer bypasses. We detect severe parsing discrepancies between the evaluated sanitizers as well as between the major browsers.

Our contributions are the following:

- `MutaGen`: A generator for HTML fragments prone to mutations during parsing.
- An analysis framework that detects diverging parsing behavior between sanitizers and web browsers.
- We then use these building blocks to assess how 11 sanitization libraries are affected by parsing differentials. We found new bypass vectors for all but two and parsing deficiencies in all of them.

The remainder of this paper is structured as follows: First, we provide a recap on the required background in Section 2. We then detail the design of `MutaGen` and our evaluation and analysis framework (Section 3), followed by an overview of our findings and the efficacy of the presented approach in Section 4. Afterward, we discuss some major takeaways and mitigation approaches (Section 5) and related work (Section 6) before we conclude in Section 7.

2. Background

In this section, we first introduce the intricacies of HTML parsing. Afterward, we discuss (mutation) cross-site scripting and how sanitization can protect against such attacks. Lastly, we showcase how parsing differentials lead to HTML sanitizer bypasses.

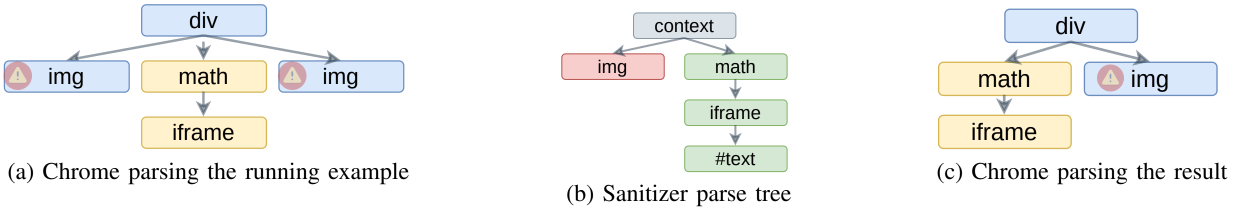
2.1. Complexities of HTML Parsing

HTML is the premiere markup language on the web, supported by all browsers. However, its evolution has not been straightforward. This is acknowledged in the official specification, which states "that many aspects of HTML appear at first glance to be nonsensical and inconsistent." [16]. Despite being seemingly simple, parsing and rendering HTML is a very involved process. From a visual point of view, one would assume that parsing HTML and XML has many commonalities. They both derive from SGML and consequently share most syntax. Modern XML parsers offer two profiles: SAX-based [17] parsing and DOM-based parsing. For the latter, the whole document is parsed into a tree structure and returned at once. SAX parsing has a lighter memory footprint as it is a stream-based parsing approach. As the parser reads the input, it emits parsing events (e.g., opening tags) as it comes across them. One would assume the same is possible for HTML, but this would be a misconception. HTML parsing is divided into two stages: tokenization, i.e., turning incoming bytes into tokens, and tree construction, which builds a Document Object Model (DOM) tree from said tokens.

A stream-based HTML parser, i.e., a parser that emits the result of each step in the tree construction stage, can never be specification-compliant. Scattered across the specification are points where the parser has to rearrange previously processed elements. For example, inside a table, if the parser encounters a tag that is not allowed to occur in this position, the *foster parenting* algorithm is invoked to rearrange the DOM and rehome the offending tag [18]. For the input, `<table><div><tbody>` a stream parser would emit the opening tags `table`, `div` and `tbody`. As the `div` tag is not a valid child of `table`, the parser invokes the foster parenting algorithm to correct the input. This results in `<div></div><table><tbody>`, i.e., it moves `div` in front of the opening `table`. Consequently, a stream parser has to invalidate already emitted events, defeating its purpose. Thus, accurately parsing HTML is only possible in a single pass. This complexity is a direct result of the desire to be always able to render a website, even if it violates the HTML specification in one way or another. Instead of rejecting invalid markup, modern browsers attempt to repair the input and display it regardless. This repair step involves the aforementioned DOM transformations, such as foster parenting, effectively mutating the input. Websites violating the specification are commonplace even today [19], preventing browser vendors from tightening the parsing process without breaking the web.

Another noteworthy aspect is that modern browsers support two HTML parsing algorithms, document and fragment parsing [20]. Document parsing is the regular parsing mode which processes a whole document. The fragment parsing mode instead relies on a context element and returns a DOM fragment, i.e., a tree of nodes rooted at the context element. It is, for example, used for `.innerHTML` assignments. Differences in the behavior of these two modes are another source of potential issues. A well-known difference between the two parsing modes is the handling of `script` tags, which are only executed in the document parsing mode. The HTML standard mandates support for both of these two parsing algorithms. However, modern browsers might implement several parsers for each parsing mode. Chromium, for example, has two fragment parsing algorithms. The *fastpath parser* is used if the fragment only contains tags that do not require DOM rearrangements, and upon encountering such a tag, it bails out to the regular one, which supports the whole tag range [21]. By not considering all the intricacies of HTML, the fastpath parser is generally faster. To top things off, HTML allows embedding so-called *foreign content* to support increasingly complex use cases. Both typesetting instructions for math formulas (via MathML [22]) and vector graphics (via SVG [23]) can be directly inserted into HTML documents. As they also share HTML's ancestry, they also share some syntactic structure (and even tag names at times), but additional complexities arise due to this combination.

Example. Consider the input from Figure 2a, which serves as the running example throughout this section. When assigning it to the `.innerHTML` attribute of a `div` element, Chromium parses the first `img` tag as an HTML element



(a) Chrome parsing the running example

(b) Sanitizer parse tree

(c) Chrome parsing the result

Figure 1: Parsing differential leading to sanitizer bypass

```
1 <img src=x onerror=f()> <math> <iframe> <img
↪   src=x onerror=f()>
```

(a) User input

```
1 <math> <iframe> <img src=x
↪   onerror=f()></iframe></math>
```

(b) Sanitized result

Figure 2: Payload before (2a) and after (2b) sanitization.

and adds it as the context nodes (i.e., the div element), first child. Then, upon encountering the math tag, the parser switches to the MathML mode (i.e., nodes are added with their namespace set to MathML) and adds math as the second child. The following iframe tag is also parsed in MathML mode and added as the first child of math. Next, the second img is processed. It is among the list of elements that cause the parser to switch back to the HTML [24]. To do so, it closes the currently open elements (i.e., iframe and math) and inserts the img tag as the context's third child, resulting in Figure 1a. HTML nodes are depicted in blue, and those in the MathML namespace are in yellow.

2.2. Cross-Site Scripting

Cross-site scripting (XSS) is the most common vulnerability class on the web. The goal behind an XSS attack is for the attacker to execute code within the security domain of the website. This allows them to exfiltrate data such as cookies or inputs, perform actions on behalf of the user, or manipulate the website's content to trick the user into performing unwanted actions. An XSS vulnerability requires the attacker to be able to control some parts of the markup of the website. Due to the fact that in HTML, there is no distinction between markup and data, at every point where user-controlled data ends up on a website, there is a potential XSS vulnerability.

Consider a website allowing users to leave comments, a basic form of community building. If a malicious user puts in the running example from Figure 2a, every other visitor's browser parses the supposed comment as in Section 2.1. The example string contains two img tags, both referencing an unavailable destination. Upon failing to load the nonexisting images, the browser executes their error handlers and calls $f()$ twice, highlighted by the warning sign in Figure 2a. The call to f happens inside the website's origin, giving the attacker complete access to each visitor's session.

2.3. Sanitization

To prevent XSS, special care is required to ensure user input is free from unwanted HTML markup. In this case, unwanted means tags executing code (such as the img tag in the example) but can also include tags changing the website's layout in an undesirable fashion. The process of removing such unwanted markup is called sanitization. To do this accurately, the sanitizer has to determine whether a specific piece of text includes markup that might execute code. A common approach to sanitization is to parse the input according to the HTML specification and to operate on the resulting DOM tree. The sanitizer then traverses the DOM and removes or transforms nodes according to, e.g., an allowed list of harmless tags or a block list of tags to remove. Afterward, the sanitized DOM is serialized back into its textual representation and returned to the caller.

For example, a sanitizer configured to allow both math and iframe tags and to remove all img tags. When processing the running example from Figure 2a, its parsing result is depicted in Figure 1b with a synthetic node as its root. To remove harmful tags, it considers each node in the tree and removes the first image node, highlighted in red. All other nodes (colored green) are in its allow list (text nodes implicitly) and, therefore, stay untouched. The serialization step again traverses the tree and converts each node to its HTML representation. Here, the input is usually cleaned beyond removing XSS payloads. As depicted in Figure 2b, the sanitizer adds closing tags that were omitted from the input.

Sanitization stands in contrast to encoding, another popular form of ensuring attacker-controlled input is free from markup. The difference is that sanitization allows certain tags to pass through and only removes (or encodes) potentially dangerous parts of the string. Encoding, on the other hand, replaces control characters with their escaped form. If a string is inserted in the HTML context, e.g., inside a div tag <div>${name}</div>, it would suffice to replace all control characters with their character references. Turning <script> into <script> would reliably prevent injection attacks in this case. Encoding should be used if the user shall not be able to influence the markup while sanitization allows the input to contain markup. They, therefore, serve different purposes. We only focus on sanitization; the security of encoding-based protection schemes is outside the scope of this work.

2.4. Mutation Cross-Site Scripting

Mutation Cross Site Scripting (mXSS) is a subclass of the generic XSS vulnerability group popularized by Heiderich et al. [25]. Such a vulnerability occurs if an HTML fragment is parsed, serialized, and yields a different result upon being parsed again. Initially, this was limited to cases where, due to updates to the DOM, the browser's HTML parser would parse an HTML fragment a second time. These vulnerabilities were based on problematic behavior of the browsers, i.e., bugs, and were resolved there.

However, over time, the vulnerability class mXSS also started encompassing what Heiderich called "mutation based attacks". Here, the initial parsing and serialization steps happen inside a sanitizer, and only the second parsing step occurs inside the browser. For such a vulnerability to manifest, the combination of HTML parsers of the sanitizer and the browser must diverge in a way that the sanitizer can be bypassed. This happens if, for example, the sanitizer parses the part of the input containing the exploit payload as part of a text node and returns it unchanged. If the browser, upon parsing the sanitizer's output, parses the assumed text content as markup, the payload is executed, introducing an XSS vulnerability.

A sanitizer affected by a parsing differential could parse the example as shown in Figure 1b. We detail the differences in the parsing and how this opens the door to a bypass in the following. The sanitizer is unaware of the namespace transition rules for foreign content and considers all elements as if they were parsed according to the HTML parsing rules. In HTML mode, everything inside the `iframe` tag is parsed as text. If the sanitizer simply echoes back text nodes, the second `img` tag passes through unmodified.

Upon parsing the output in Chromium, the `iframe` tag is parsed as a custom MathML tag, and when encountering the `img` tag, the parser switches back to HTML mode, closing all open MathML tags in the process. The XSS payload is thus lifted out of the `iframe` and moved as a direct child of the context element, causing code execution upon evaluation, shown in Figure 1c. Thus, mutation-based bypasses are possible whenever there is a difference in parsing behavior between the browser and sanitizer. These kinds of bypasses are the focus of this work.

3. Uncovering Parsing Differentials

To detect parsing differentials and mutation-based sanitizer bypasses, we built a testing framework consisting of three stages: Input generation, sanitization, and evaluation. The framework is depicted in Figure 3. We made the source code for the testing framework, i.e., `MutaGen` and the testbed, available online [26]. We now first detail the results of analyzing the HTML specification and then detail each stage of our testing framework in the following.

3.1. HTML Analysis

With the goal of generating mutation-prone HTML fragments in mind, we first analyzed the HTML specification

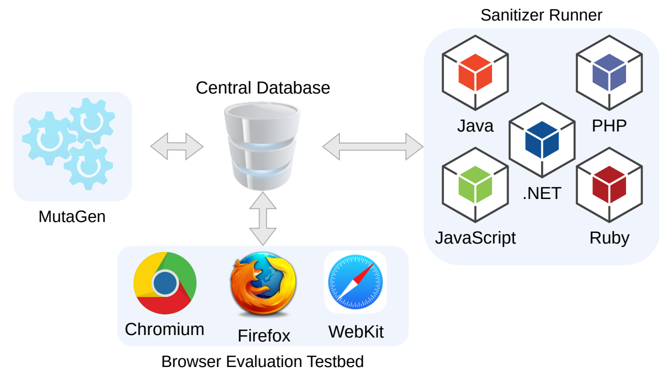


Figure 3: Sanitizer Evaluation Setup

as well as past sanitizer bypasses based on parsing differentials [27–30]. Based on inspecting the HTML element [31] semantics and their corresponding parsing specifications [32], we collected elements with complex parsing rules. The element specification provides a general description of all elements, including restrictions on where they can occur, whether closing tags can be omitted, and their content model. The content model of an element specifies what other elements are allowed as its children. The parsing specification, on the other hand, describes how the parser constructs the DOM tree.

An example of a tag with complex parsing rules is the `iframe` tag. It is noteworthy as its element specification and parsing specification disagree. Its content model is *nothing* [33], stating the element "must contain no Text and no element nodes" [34] but the parsing specification instructs to parse its content as text, directly violating the content model. We identified a total of 47 tags, which can be divided into the following groups of elements: 1) Those with restrictions on their content (e.g., `select` can only contain specific child elements) 2) restrictions on where they can occur (e.g., `tr` can only occur inside a `table`) 3) constraints on how often they can occur (e.g., there can only be one `title` while `forms` can not be nested) 4) with disagreements between parsing and element specification (e.g., `iframe`) 5) causing namespace transitions (e.g., `svg` or `math`) 6) and lastly those that are deprecated (e.g., `xmp`, which used to display HTML code without executing it) The full list of tags with reasoning for their selection is provided in Table 7.

The parsing specification contains a "parse errors" [35] section, which is an additional source of parsing quirks we identified as potentially challenging to implement. While the specification explicitly allows a parser to abort the parsing process upon encountering such an error, no parser does this. Instead, they emit erroneous output or rewrite the input. The identified quirks include 1) incorrect comments 2) invalid attributes 3) attributes inside closing tags.

These identified complexities are the foundation for our generation approach.

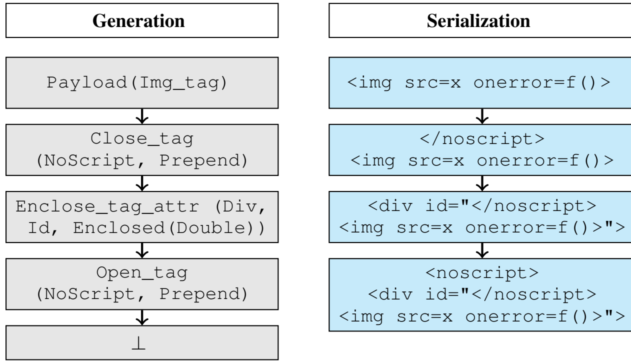


Figure 4: Simplified Payload Generation and Serialization.

3.2. `MutaGen`: HTML Fragment Generator

The basic idea behind `MutaGen` is to approach the generation process iteratively. We first select an initial payload $\mathcal{P}$, i.e., a piece of HTML triggering JavaScript execution, and subsequently extend $\mathcal{P}$ with surrounding HTML structure. The initial payload is as basic as possible by design. Generally, two kinds of injection vectors lead to XSS: tag-based and attribute-based injections. Consequently, we chose two payloads (i.e., `script` and `img` tags) to represent these categories. These are the most well-known payloads for their respective categories. Hence, we expect every sanitizer to handle them. During the HTML analysis, we noticed that the specification instructs parsers to rewrite `image` to `img` tags. This behavior represents a third class, parsing quirks, and thus, we added `image` to the set of initial payloads to cover this class of behaviors as well. While more advanced payloads may uncover additional bypasses, detecting vulnerabilities due to, e.g., a sanitizer missing specific event handlers in a block list was not the focus of this work.

Once an initial payload is selected, `MutaGen` randomly selects transformations which, when applied to the current payload, modify it. An example of such a transformation is to prepend an opening tag such as `div`, i.e., transforming $\mathcal{P}$ into $<div>\mathcal{P}$. Upon reaching a predefined limit on the number of transformations (set to 25 for our experiment) or selecting the termination transformation (denoted as $\perp$), the generation is complete. The $\perp$ transformation allows us to generate payloads of varying length, as always applying 25 transformations results in payloads of uniform length. We then check that the generated payload is unique, i.e., has not been generated by a prior run, and that it is not entirely made up of whitespace or closing tags. Such a payload can never cause interesting behavior, as closing tags without opening tags are discarded. If both conditions hold, we serialize it to its string representation and store both its abstract as well as its textual representation in a central database.

This approach allows us to trivially add transformations that alter the whole accumulated payload, e.g., to perform XML encoding. We implemented the HTML fragment generator in slightly over 1,100 lines of OCaml code; it manipulates payloads with 23 transformations, most of them

Table 1: Examined Sanitizing Libraries

Name	Version	Total Downloads	Language	Vulns.
DOMPurify (*)	2.3.10	399,001,216	JavaScript ‡	1
	3.0.3			1
sanitizer	0.1.3	41,063,147		†
google-caja-sanitizer	1.0.4	242,850		
sanitize-html	2.7.0	276,882,692		0
HtmlSanitizer	8.0.601	19,800,000	.NET	2
HtmlRuleSanitizer	1.6.0.1	306,100		2
Typo3 html-sanitizer	2.0.15	1,950,185	PHP	4
rgrove/sanitize	6.0.0	60,928,006	Ruby	1
loofah	2.21.3	396,621,861		0
AntiSamy	1.7.3	No data available	Java	3
JSoup	1.16.1			2

*: jsdom version 19 and 22, †: Based on the same code base, both abandoned; therefore vulnerabilities not broken down, ‡: Retrieved with https://npm-stat.com

parameterized. For example, the `Enclose_tag_attr` transformation in Figure 4 is parameterized over the tag, the attribute's key, and quotes. The full list is provided in Table 6 and their parameters in Section A.1.

Example. One HTML parsing aspect we discovered as problematic for most sanitizers is correctly terminating `noscript` tags. Figure 4 details a simplified generation run yielding a payload capable of generating a payload that bypasses several sanitizers. On the *Generation* side in Figure 4, a list of transformations is created, starting from an initial payload, here an `img` tag. With each subsequent transformation, `MutaGen` adds surrounding structure to the payload. First, it prepends a closing `noscript` tag and then encloses the accumulated payload inside the double-quoted `id` attribute of a `div` tag. Next, an opening `noscript` tag is prepended again, and the generation terminates with the $\perp$ transformation. This yields the list of transformations given on the *Generation* side in Figure 4 top to bottom. To hand this sample to a sanitizer, it first has to be serialized into HTML code. Each step of this process is shown on the right side (captioned *Serialization*) of Figure 4.

3.3. Payload Sanitization

For each generated fragment, we now want to analyze how different sanitizers process it. We selected the sanitizers in our testbed by searching the package repositories of JavaScript, .NET, Ruby, PHP, and Java for popular server-side HTML sanitizers. We then inspected their source code to determine whether they use an HTML parser that we can access to retrieve its internal state.

Using an actual HTML parser is a necessary prerequisite to be affected by parsing differentials, i.e., to be in scope for our work. Therefore, we did not include any sanitizer that simply cleans the input based on, e.g., regular expressions. Attempting to process HTML via regular expressions is problematic in its own right but not the focus of this work. We refer the reader to [1, 9, 36–38] for security

assessment of such sanitization approaches. This allows us to focus on detecting HTML parsing divergences and their effects on sanitizers. To perform a meaningful analysis of different parsing behaviors, we also require access to their internal state. That is, how did the underlying HTML parser understand the input the sanitizer attempts to clean? This internal parsing state is not made public in any of the considered sanitizers. Therefore, we added functionality to extract it. This was either done by setting appropriate hooks, e.g., for DOMPurify, or by modifying the code, e.g., for Google Caja-based ones, while keeping the sanitization logic untouched. Thus, for every sanitizer invocation, we store a DOM-like structure (representing the sanitizer's internal state) together with the sanitizer result. This allows us a meaningful comparison between sanitizers. This resulted in 11 sanitizing libraries across 5 programming languages. Their exact version numbers as well as additional meta information, are detailed in Table 1.

3.3.1. Sanitizer Configuration. Most of the tested sanitizers allow for a wide range of configuration options. Those usually include allowing or restricting additional tags, restricting which attributes are allowed, and so on.

We tested each sanitizer in its default configuration but also considered a more lenient variant, explicitly allowing all tags and attributes generated by our tool if such a customization is possible. loofah, a sanitizer for Ruby, or both Caja-based ones do not allow for such customizations. Consequently, they are only tested in the default configuration.

We did not attempt to enforce misconfigurations. One sanitizer in our test set, namely sanitize-html, requires setting an aptly named flag (called `allowVulnerableTags`) to enable some tags generated by `MutaGen`. We did not set these, as the documentation clearly states that setting them renders the sanitizer pointless. Instead, we limited ourselves to allowing tags via the regular mechanisms.

Each generated payload was consequently sanitized by every sanitizer from Table 1 in both their default and relaxed configuration. Their outputs were inspected to check whether they still contained a call to our reporting function, and if that was the case, they were marked for evaluation. In addition, every generated payload was also marked for evaluation without sanitizing it first.

3.4. Payload Evaluation

While the sanitizer's parsing state is sufficient to deduce parsing differentials between sanitizers, finding bypasses requires evaluating the output in a real browser. To do this, we leveraged the browser automation framework Playwright in version 1.27.0. It automates running Chromium, Firefox, and WebKit in versions 107.0.5304.18, 105.0.1, and 16.0, respectively. Our framework evaluates each sample marked for evaluation in each browser and parsing mode combination. That is, to ensure both document and fragment parsing modes are evaluated, each marked sample is evaluated twice. For fragment parsing, we assign the payload to `innerHTML` of the document's `body` element, while for document parsing, we directly insert it into the `body` of the page. This allows us to detect differentials between the parsing behavior of the two algorithms or bypasses that only manifest in either of them.

As modern web browsers are highly complex pieces of software, the evaluation step is rather time-consuming. To ensure that – even under heavy system load – we do not miss any calls to the reporting function, we waited for 75 ms after inserting the payload into the page. Together with the surrounding setup code, such as opening a new page inside the browser, evaluating a single payload took about 90 ms.

Table 2: Number of Evaluated and Executed Samples

Sanitizer	Evaluated		JS Executions	
	Default	Lax	Default	Lax
None	12,000,000		855,290	
DOMPurify	1,770,812	2,210,713	0	341
DOMPurify (jsdom19)	1,518,562	1,716,177	31	154
sanitizer	2,721,962		4,971	
google-caja-sanitizer	2,866,299		5,354	
sanitize-html	1,347,494	4,330,265	0	0
HtmlSanitizer	7,512,576	7,652,333	0	966
HtmlRuleSanitizer	607,496	7,269,990	5,080	34,384
Typo3	11,705,381	11,710,159	4,754	52,214
rgrove/sanitize	1,816,383	4,988,545	0	2,178
loofah	4,452,547		0	0
AntiSamy	5,473,627	6,696,708	7	2,116
JSoup	5,970,206	8,132,379	0	13,265

4. Parsing Differentials: Prevalence and Impact

We generated 12 million unique payloads for this study. The generation, sanitization, and evaluation pipeline took 14.5 days in total, running concurrently on a server powered by an AMD EPYC 7702P 64-Core CPU and 512GB of main memory. During the evaluation, each call to the reporting function from our payloads was recorded, and the corresponding sample was marked as causing code execution. The total numbers of samples marked for evaluation and samples causing JavaScript execution per sanitizer are provided in Table 2.

The number of evaluated samples already gives a hint about different strategies employed to clean input. Sanitizers with few evaluations (e.g., sanitize-html or DOMPurify) remove problematic parts, while others, such as the Typo3 sanitizer tend to keep the basic structure in place. An example to showcase this behavior is the payload <textarea><script>f(). One strategy is to delete the content of `textarea`, e.g., employed by sanitize-html, which in turn deletes the call to our reporting function, $f()$. A second strategy, for example used by DOMPurify, is to encode the content of `textarea`, i.e., turning <script>f() into <script>f(). Both approaches prevent the execution of the XSS trigger but have tradeoffs in terms of usability. Any benign content of such a `textarea` tag is

equally deleted when applying the first strategy. There is, however, no correlation between employing either strategy and being more susceptible to bypasses. HtmlRuleSanitizer, sanitizer and google-caja-sanitizer are among those with the fewest evaluated samples in their default configurations but have the most samples with JavaScript execution.

Please note that payloads causing JavaScript execution after sanitization are not a direct subset of those executing JavaScript without sanitization. In total, 875,133 payloads were executed at least in one configuration. Without applying sanitization first, 855,290 payloads did cause JavaScript execution. This means that 19,843 payloads did not execute on their own but required the sanitization step to turn them from a benign into a dangerous payload.

One would expect the number of executed payloads to be equal across browsers. This is not the case. Chromium executed 862,780 and 668,897 in document and fragment parsing mode, respectively, the numbers are fairly similar for WebKit with 863,071 and 668,893 executions. Both browsers originate from the same code base, so similar behavior is expected. For Firefox, however, the results are significantly different. It executed 858,523 payloads in document and only 497,941 payloads in fragment parsing mode. The reasoning for this significantly different number of executed payloads rests in a deviation from the specification for Firefox, which we detail in Section 4.4.

Note that the number of executed samples for fragment parsing is lower across the board. This is expected, as payloads using `script` tags as code execution triggers never execute in fragment parsing mode.

All payloads that executed JavaScript despite having been sanitized were marked as bypasses and consequently analyzed. We filtered them for common root causes (i.e., two payloads containing the same issue and different surrounding markup) and disclosed the vulnerability to the respective maintainers. This was greatly aided by us storing the internal parsing result of each sanitizer, as it allows us to quickly asses what root causes led to the bypass. All bypasses found over the course of this study are summarized in Table 3. We did not break down the issues found in the two Caja-based sanitizers for brevity, as they are both unmaintained.

We were able to bypass all evaluated sanitizers except sanitize-html and loofah. 6 out of 11 sanitizers were affected in the default configuration, which tends to be rather restrictive. For three additional sanitizers, we only found bypasses in the more permissive configuration. However, due to each website having unique needs in terms of tags to allow, we assume that adjusting the default configuration is commonplace. This can be seen when looking at libraries such as AntiSamy, which ships with configurations taken from popular websites such as Slashdot or eBay. The provided configurations contain very different allow lists, with the eBay one, for example, being very permissive, even allowing tags such as `noscript`.

While the relaxed configuration set by us is extremely permissive, all bypasses found by us usually only require adding one or two tags to the allow list, i.e., only a subset is needed. Testing these different subsets independently, however, would lead to an infeasible number of payloads to evaluate. Therefore, we set a very permissive configuration in which we minimized the changes required for the specific bypass before reporting them.

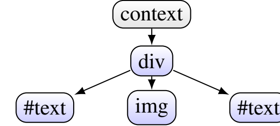


Figure 5: DOM structure of `<div>HT<img>ML`

4.1. Prevalence of Parsing Differentials

The reason for using an HTML sanitizer is to allow the user to preserve some form of user-provided markup. Suppose one wants to ensure input does not influence the website's markup at all. In that case, the safe way is to simply encode the input (cf. Section 2.3), ensuring only text content ends up in the final document. Therefore, we assume that users of these sanitizers expect them to remove only the actual XSS trigger and other forbidden elements while preserving benign HTML structures as is. To do this, the sanitizer's parsing result has to be as close to the browser's as possible. Otherwise, benign parts of the DOM might get removed, degrading the website's functionality.

To assess the similarity between parsing results, we first select a metric to compare DOM trees.

4.1.1. Bag of XPaths Similarity Score. The *Bag of XPaths* metric [39] is one way to calculate the similarity between two websites, i.e., DOM trees. Here, each document is converted into a set of XML Path Language (XPath) expressions, one for each leaf node in the DOM. For example the fragment `<div>HT<img>ML` has the DOM structure pictured in Figure 5 and is converted into three XPath expressions: `/div[0]/text[0]`, `/div[0]/img[0]`, and, `/div[0]/text[1]`. To calculate the similarity between two documents D_1 and D_2, we first compute the set of XPaths for both, resulting in n_1 and n_2, respectively. We then take the intersection of n_1 and n_2 to compute c and apply Equation (1).

$$similarity(D_1, D_2) = \frac{|c|}{|n_1| + |n_2| - |c|} \tag{1}$$

If two documents share no common XPaths, their similarity is 0, and if they have exactly the same set of XPaths, i.e., their DOM trees are equal, the result is 1.0. We have slightly adapted the metric to better fit our setting. Compared to the original implementation of this metric, we omitted the notion of generalized XPaths, which are supposed to express repeating patterns. Such patterns are very likely to occur on actual websites, e.g., multiple rows of a table all have the same structure. `MutaGen`, however, does not generate such structured markup. Therefore, generalized XPaths might, at best, introduce noise in our case, as the generalization would detect patterns where there are none. Additionally, we

Table 3: Sanitizer Bypasses Found with MutaGen

Id	Sanitizer name	Config.	Cause	Description	Status
	google-caja-sanitizer (*) sanitizer (*)	Default		Various	Abandoned Projects
1	DOMPurify (jsdom 19)	Default	SI 6	Decodes and reflects text content	Independently fixed
2	DOMPurify	Relaxed	PI 1	`noframes` not parsed correctly	Resolved
3	Typo3	Default	PI 4	CDATA sections not parsed correctly	2022-23499 ‡
4	Typo3	Default	PI 5	Closing bang comment not detected	2022-36020
5	Typo3	Relaxed	PI 1	Namespace confusion	2022-23499 ‡
6	Typo3	Relaxed	PI 2	`noscript` content parsed as HTML instead of as text	2023-38500
7	AntiSamy	Default	†	Tags not listed in the configuration not handled securely	Acknowledged
8	AntiSamy	Relaxed	PI 5	Closing bang comment not detected	Acknowledged
9	AntiSamy	Relaxed	PI 1	Tags with text content are not closed if they contain a comment	2023-43643
10	HtmlRuleSanitizer	Default	PI 5	Closing bang comment not detected	Resolved
11	HtmlRuleSanitizer	Relaxed	PI 1	Wrong parsing of tags with text content allows to break out of attributes	Reported
12	HtmlSanitizer	Relaxed	PI 2	`noscript` content parsed as markup.	Resolved
13	HtmlSanitizer	Relaxed	PI 3	Firefox parsing differential	Acknowledged
14	rgrove/sanitize	Relaxed	PI 2	`noscript` content parsed as markup instead of as text	2023-23627
15	JSoup	Relaxed	PI 3	Namespace confusion	Resolved
16	JSoup	Relaxed	PI 2	`noscript` content parsed as markup instead of as text	Resolved

†: Logic bug. *: Based on the same code base, largely affected by the same vulnerabilities. ‡: Two separated vulnerabilities got grouped into this CVE.

added the notion of text nodes. The original metric is only concerned with the relationship between tags. However, if text nodes are moved from one tag to a different one during sanitization, this has a profound impact on the rendering of the resulting fragment. Thus, we decided to add text nodes as well. The same applies to comments and CDATA sections if the parser recognizes those. While they do not influence the rendering, parsing them incorrectly leads to a different result upon serialization. To model this influence, we also add XPaths for text, comment, and CDATA nodes, as they are always leaf nodes.

Table 4: Similarity of Sanitizers and Browsers Parse Tree.

Sanitizer	Chrome		Webkit		Firefox	
	F	D	F	D	F	D
DOMPurify	0.87	0.87	0.87	0.87	0.81	0.86
DOMPurify (jsdom19)	0.88	0.88	0.88	0.88	0.82	0.88
sanitizer	0.36	0.36	0.36	0.36	0.37	0.36
google-caja-sanitizer	0.50	0.50	0.50	0.50	0.50	0.50
sanitize-html	0.39	0.39	0.39	0.39	0.41	0.39
HtmlSanitizer	0.90	0.90	0.90	0.90	0.84	0.90
HtmlRuleSanitizer	0.15	0.15	0.15	0.15	0.15	0.15
Typo3	0.52	0.52	0.52	0.52	0.53	0.52
rgrove/sanitize	0.94	0.94	0.94	0.94	0.88	0.94
loofah	0.22	0.22	0.22	0.22	0.25	0.22
AntiSamy	0.58	0.58	0.58	0.58	0.58	0.58
JSoup	0.51	0.51	0.51	0.51	0.52	0.51

F: fragment parsing, D: document parsing

4.2. Parsing Accuracy

We calculate this by retrieving the resulting DOM trees after rendering each unsanitized payload in all browsers and configurations and comparing them to the internal representation of the sanitizer. Due to implementation differences, these internal DOM-like structures can look fairly different. DOMPurify, for example, creates a complete HTML document with head and body sections, while others operate on a document fragment. Thus, we first unify the internal representations to all have the same shape. The results are provided in Table 4. If the sanitizer's HTML parser would perfectly match the browser's, the similarity score would be 1.0. A score of below 0.5, on the other hand, means that for two DOM trees, more than half of their leaf nodes only occur in either DOM tree. That is, they differ by a significant amount.

As the table shows, the similarity scores vary greatly between sanitizers. While some (e.g., DOMPurify, HtmlSanitizer or rgrove/sanitize) are operating on a fairly accurate internal structure, others such as HtmlRuleSanitizer produce wildly different parsing results.

Interesting to note is that while the similarity of fragment and document parsing modes are very similar for Chromium and WebKit, the scores for Firefox diverge noticeably. This is a result of the Firefox fragment parser deviating from the specification, which we discuss in depth later on.

4.3. Classifying Parsing Deficiencies

As shown previously, the different parsers do not always accurately parse their inputs, compared to the major browsers. Having access to the sanitizer's internal representation allows us to also analyze where their HTML parsers violate the specification. Such violations do not necessarily imply a security issue but, especially when several can be combined, are often building blocks for bypasses. In any case, they are functional deficiencies, frequently manifesting as overzealous transformations of the output.

4.3.1. Parsing. We found five distinct parsing issues (PI), each affecting one or more different sanitizers.

1: Incorrect Parsing of Tags with Text Content Several tags instruct the parser to switch to parsing modes recognizing textual content such as RCDATA [40]. In the RCDATA state, the parser interprets everything between the opening tag until a matching closing tag as text, decoding character references in the process. If the parser does not model these

transitions, it parses the text content as if it were HTML markup. This can allow an attacker to trick the parser into parsing regular markup as if it were an attribute. Consider the string: `<iframe><div id='</iframe>'>`. Upon encountering an opening `iframe` tag, the parser switches to the RCDATA state, everything up until the closing `iframe` tag is parsed as text and added as a text node below the `iframe` node. If the sanitizer does not model this transition from HTML parsing to text parsing, it would parse the string as if the `iframe` had a `div` node with an `id` attribute containing the string `</iframe>` as its child. Then, the parser continues to look for further child elements of the `iframe` node until a top-level closing `iframe` tag occurs. Effectively, the parser attaches content that should be outside of the `iframe` tag as its children. This problem class affects all tags that have textual content, namely `textarea`, `xmp`, `noframes`, `noembed`, `iframe`, `title`, `style` and `plaintext`.

One possibility for why this error occurs is using a regular XML parser to parse HTML documents, as XML does not have such transitions. This problem only applies to tags in the HTML namespace; if, e.g., a `xmp` tag was parsed as SVG, it would have regular content. As sanitizers do not tend to make namespace information (if they are aware of it in the first place) available, we automatically labeled their DOM trees with the namespaces based on the rules for namespace transitions from the parsing specification [24].

The detection approach for this issue class works as follows: Examine the children of all nodes, which, according to the specification, shall only have text children. If at least one child is not a text node, the parser is affected by PI 1.

2: Incorrect Parsing of `noscript` This case is a special case of PI 1, but due to additional complexities, it is listed separately. The `noscript` tag has unusual parsing semantics, even for the convoluted HTML specification. Its semantics rely on a *parsing state flag*, the *scripting flag* [41], which signals JavaScript support. In the case of JavaScript support, the content of `noscript` shall be parsed as text, otherwise as markup. This feature was used to provide fallback solutions to legacy browsers without JavaScript support. While such browsers do exist, they are outside the threat model of XSS attacks. A sanitizer parsing `noscript` as if no JavaScript support was available is at risk for bypasses. This class can be detected in the same fashion as PI 1.

3: Foreign Content and Namespace Transitions When parsing foreign content, i.e., SVG or MathML segments, several integration points are available to switch the parser back to HTML mode. For example, via the `foreignObject` tag, a piece of HTML can be embedded into an SVG graphic, allowing the reuse of CSS styles. Similar integration points exist for MathML, e.g., `mtext`. It is important to note that they integrate an HTML block into the foreign content.

A number of HTML tags also have special meaning inside foreign content [24]. Instead of a seamless integration, they however instruct the parser to close the currently open non-HTML elements. As an example, consider `<svg><desc><div>X</div></desc><img>`. Consequently, the parsing result is `<svg><desc><div>X</div></desc></svg><img>`. Both `div` and `img` are among the tags terminating foreign content. The `desc` tag, however, serves as an HTML integration point, allowing the `div` tag to be part of the `svg` block. Meanwhile, upon encountering the `img` tag without such a preceding integration point, the parser closes all open SVG tags and attaches the `img` directly to the parent node. To correctly model the behavior of each tag, the parser has to be aware of the tag's namespace, and as such, it has to model these namespace transitions. Failing to do so, e.g., by attaching the `img` tag as a child of the `svg` element, falls into this category.

We detect this by first assigning namespace labels according to the specification. This allows us to scan the DOM for invalid states, such as an `img` tag as a child of a `svg` tag.

4: Incorrect CDATA Handling XML documents allow enclosing content that shall be interpreted literally and not parsed as markup in so-called CDATA sections. It can be used to represent text containing special characters or XML syntax without additional escaping. A CDATA section is written as follows: `<![CDATA[<b> to emphasize]]>`.

While HTML is derived from SGML, the parser treats CDATA sections outside of foreign content as errors. As HTML parsing never fails, it also specifies how erroneous CDATA sections shall be handled: the opening `[CDATA]` and closing `]]>` strings shall be treated as comments [42]. This handling, however, is rather unintuitive. `<![CDATA[a<b]]>` is treated as `<!--[CDATA[a<b]]-->`, matching the specification. However, if the CDATA section does contain a closing angle bracket, the resulting comment terminates early. The input `<![CDATA[<b><t>)]]>` is parsed as `<!--[CDATA[<b--><t>)]]>`, with the `t` tag outside of the comment and part of the regular DOM. If a parser expects the CDATA section as a whole to be treated as a comment, it is at risk for a bypass based on the second example. If the tag `t` was an XSS payload instead, the parser would see the payload as part of a comment and thus harmless. If a CDATA node containing one or more closing angle brackets is returned in the DOM, we mark the sample as causing PI 4.

5: Closing Bang Comments HTML specifies the syntax for comments as: `<!-- content -->`. However, it also accepts incorrectly closed comments, that is, comments closed with `--!>` [43]. If an HTML parser misses this detail, it would treat a string such as `<!-- c--!><t>-->` as if the `t` tag was inside the comment. This allows the smuggling of XSS payloads through comments if they are included in the output. We detect this issue by scanning the DOM for comments containing the string `--!>`.

4.3.2. Serialization.

To return the sanitized result to the caller, the sanitizer has to turn the internal representation back into its textual form, called serialization. This section is concerned with problematic implementations of the serialization step. The serialization usually is implemented in the HTML parser, but if it does not handle these aspects

securely, the sanitizer should take care of them to avoid easy bypasses. We derived two categories of serialization issues (SI) the bypasses are based on.

6: Decodes Text Values The HTML specification instructs the parser to decode character references. Character references have the form of e.g., `<` to encode <. To render a document, a browser has to decode such character references, as is mandated by the specification. However, if a sanitizer decodes character references and does not encode them again during serialization, there is potential to make the sanitizer turn benign input into dangerous output. This issue can occur in several parts of the DOM, namely inside text nodes, attributes, or comments.

Based on the abstract representation of the generated payload, we can easily derive which encodings were applied to the XSS trigger. If at least one encoding was applied and the decoded payload can be found inside one of the named node types, the sanitizer is affected by PI 6.

7: Failure to Encode Text Values Nodes parsed as text that the sanitizer does not encode during serialization are a significant risk for bypasses. If there is a parsing differential between the sanitizer and the users' browser, the assumed text node might be parsed as markup and a trivial bypass occurs. An example of how this can occur is `<select> <iframe><script>f()`, one of the bypasses affecting both Caja-based sanitizers. According to the specification, the content of `iframe` tags shall be parsed as text. Consequently, `<script>f()` would be seen as benign content and attached as a text node below it. However, when a browser parses the whole fragment, it behaves differently. An `iframe` tag violates `select`'s content model. The `select` tag can only contain `option`, `optgroup`, `hr` tags and "script supporting elements" [44]. Script supporting elements include `script` and `template` tags. Consequently, an `iframe` is not a valid child of `select`, and the browser drops it during tree construction. This turns the supposedly harmless text node into markup that is regularly parsed, and the `script` is finally executed.

To defend against such attacks, a sanitizer would have to consequently encode all text nodes and attribute values. Performing such encoding would have prevented all bypasses from Table 3 but bypass 1 and 7. We detect missed encoding steps by checking if the XSS trigger is located inside a text node or attribute value in the sanitizer's internal DOM and whether it occurs in the output in unencoded form.

4.3.3. Affected Sanitizers.

Table 5 breaks down what sanitizers are affected by which parsing or serialization issues. In summary, we detected functional deficiencies in every analyzed parser and problematic handling of text values in all but two. The two sanitizers not affected by either serialization issue, i.e., those that do not remove encodings from their input and consequently encode text nodes, are the two where we found no bypasses.

The fact that each parser is at least affected by two parsing issues is cause for concern and highlights the complexity of the parsing task.

Table 5: Parsing and Handling Issues Affecting Each Sanitizer

Sanitizer	Parsing					Serialization	
	PI 1	PI 2	PI 3	PI 4	PI 5	SI 6	SI 7
AntiSamy	●	●	●	●	●	●	●
sanitizer	●	○	●	○	○	○	●
google-caja-sanitizer	●	○	●	○	○	○	●
DOMPurify	●	●	●	○	○	●	◐
DOMPurify (jsdom19)	●	●	●	○	○	●	●
HtmlSanitizer	●	●	●	○	○	◐	◐
HtmlRuleSanitizer	●	●	●	○	●	○	●
JSoup	●	●	●	●	○	◐	◐
loofah	●	●	●	○	○	○	○
sanitize	●	●	◑	○	○	●	◐
sanitize-html	●	●	●	○	○	○	○
Typo3	●	●	●	●	●	●	○

●: Affected, ◐: Affected in relaxed configuration, ○: Unaffected,
◑: Affected but not in scope of threat model

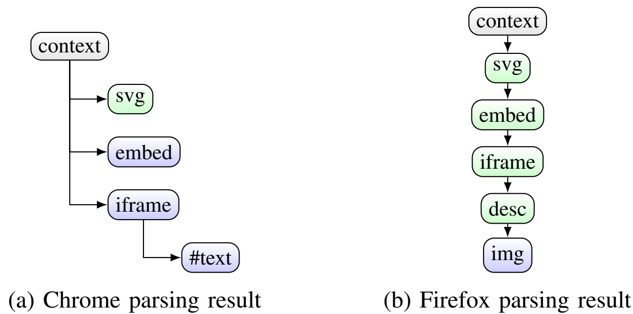
The first problematic aspect is the correct parsing of tags with textual content. Every analyzed parser fails at this task for at least some samples. Similarly, the handling of `noscript`, which not only requires a parsing transition but also relies on runtime information in the browser, is a frequent source of mistakes. How HTML parsers implement this aspect differs, with some requiring users to pick a value for the scripting flag, e.g., as AngleSharp for .NET. Others, such as the Nokogiri HTML parser for Ruby, do not offer a choice at all. The sensitive default for sanitization code would be to default to scripting being active. Only the Google Caja-based sanitizers had this setting, however.

If the parser is mainly used for tasks such as web scraping, defaulting to false seems sensible. It is, however, a potential security issue, as bypasses 6, 16 and 14 show. This quirk received considerable media attention in 2019 when Masato Kinugawa found a bypass in the Google Search Bar [45] based on the duality of `noscript`. Nevertheless, as our results show, this has not led to awareness for authors of sanitizing libraries.

Foreign content (PI 3) is similarly a common source of mistakes. The rules on when to switch namespaces are not correctly implemented in any analyzed sanitizer. All sanitizers we were able to bypass are also affected by at least one serialization issue, as those bypasses usually rely on a parsing mistake combined with a lack of encoding to succeed. Interestingly, HtmlRuleSanitizer allows the user to configure if HTML entities in text nodes shall be encoded. Giving control to the user might seem desirable, but without additional warning, enabling this option allows to trivially bypass the sanitizer.

4.4. Browser Parsing Differentials

Another issue for authors of sanitization routines is the aspect that browsers might diverge from the HTML specification in some cases. Firefox's fragment parser, for



(a) Chrome parsing result (b) Firefox parsing result

Figure 6: Parsing differential between Chrome and Firefox. Blue nodes have the HTML namespace, green ones SVG.

example, does not parse foreign content correctly, i.e., it is affected by PI 3. Instead of closing the foreign namespace upon encountering an HTML tag supposed to terminate foreign content, it stays in the current parsing mode. Normally, this simply results in a website being rendered incorrectly. However, such differences can be abused to bypass sanitizers as well. A payload exemplifying this issue is <svg><embed><iframe><desc><img src=x onerror=f()>. The parsing results for Chromium (Figure 6a) and Firefox (Figure 6b) are provided in Figure 6. Chromium terminates the SVG context upon encountering the embed tag and parses the remaining input as HTML. Therefore, the opening desc and the image tag are parsed as text and attached under the iframe node, preventing the execution of the error handler. Firefox, on the other hand, parses both embed and iframe as SVG tags, causing them to lose their HTML semantics. Then, upon encountering desc, the parsing rules for SVG apply, and the parser switches back to HTML [46]. Consequently, Firefox parses the img tag as a regular HTML tag and executes its onerror handler, calling f. A sanitizing routine purely relying on the specification to assess whether a tag needs sanitization is, therefore, vulnerable to bypasses such as the one described here. Thus, to accurately sanitize input, a sanitizer either has to be aware of all possible browser quirks or put users of selected browsers at risk. Without information about the browser of the specific user, it then has to find the lowest common denominator, degrading its output. We found the example provided above during our study affecting HtmlSanitizer with a relaxed configuration (bypass 13).

Resolving this issue has proven to be involved, as it is unclear who is responsible for fixing such bugs. A sanitizer adding a workaround for a browser bug would degrade the output for compliant browsers. Not fixing it, however, leaves users of non-compliant browsers at risk.

We have reported this parsing differential to Mozilla, and it awaits resolution at this time. Please note that this example also manifests in a more involved form. For example, we detected payloads for JSoup where this difference allows lifting the payload from an attribute value.

5. Discussion

The results presented in the previous section paint a dire picture of the state of server-side HTML sanitization, directly answering the initial questions. Due to the lack of information available to the sanitizers, it is not feasible to build one that is both accurate and secure, and popular sanitizers fall well short of this goal.

We now discuss some problematic aspects in depth, detail the disclosure process, explain how to mitigate XSS vulnerabilities in the presence of parsing differentials and finally provide a general outlook.

5.1. Foreign Content

The fact that HTML allows embedding foreign content, i.e., SVG or MathML snippets, adds significant difficulties for authors of parsing and sanitization libraries. As every namespace transition changes the semantics of several tags, missing even a single one is often enough to introduce a vulnerability. As shown in the previous Section, none of the tested sanitizers implement this correctly, and even the major browsers do not always get it right. This makes the question of how sanitizers should handle such mixed documents an interesting one. rgrove/sanitize deviates from the remaining libraries, as it explicitly warns that it does not support sanitization of foreign content. It defaults to simply removing everything it parses as foreign content, which frequently includes regular HTML content due to not implementing the complex namespace transition rules. This warning is not enforced in the library itself, as it is possible to add the offending tags to its allow list without further warning. We have reported issues related to incorrectly parsing foreign content to rgrove/sanitize's maintainers, and they added additional protection mechanisms, such as always escaping the content of text nodes.

5.2. Weaponizing Sanitizers

Surprisingly, in some cases, the sanitizer turned initially harmless HTML fragments into a dangerous payload. Such cases occur if the sanitizer relies on the underlying parser's serialization functionality. DOMPurify, using jsdom $v19$, was affected by such an issue, namely bypass 1. When sanitizing <svg><style><img src=x onerror=f()><keygen> the sanitizer recognized the escaped img tag as harmless text. It then returned the string <svg><style><img src=x onerror=f()> which is clearly problematic. During serialization, the XML encoded text node, i.e., the img tag, got decoded, which armed the payload. The presence of a trailing void (i.e., self-closing) element caused jsdom to XML decode the text node, which was then picked up by the browser's DOM parser. This validates the inclusion of destructive transformations, such as encoding operations, for our payload generation. URI encoding, on the other hand, was never reverted by any tested sanitizer.

5.3. Disclosure Process

We divided the disclosure process into two parts: vulnerabilities and functional deficiencies. Each sanitizer bypass puts a considerable amount of website operators at risk of exploitation and, consequently should be resolved quickly. All vulnerabilities stemming from parsing differentials can be prevented without solving the underlying issue. This usually requires degrading the output quality but might be an attractive short-term solution. Resolving parsing issues such as PI 3 or PI 1, on the other hand, often requires fundamental reengineering of the parser itself. We are currently working on reporting the parsing issues discussed in Section 4.3.3 as well as more basic parsing errors we uncovered to their respective maintainers. HtmlRuleSanitizer for example parses the input `<div id= <div>` as `<div id=""><div>`. This behavior does not follow the specification, which mandates it to be parsed as `<div id="<div/">`.

Vulnerability Disclosure. We contacted the corresponding maintainers of all actively maintained libraries from the test set regarding our findings. At the time of writing, most of them have been fixed, as shown in the Status column in Table 3.

As the main focus of DOMPurify [47] lies on client-side usage, using it on the server is more involved. Here, it relies on an external HTML parsing library to produce a DOM tree, with the manual recommending jsdom. The chosen HTML parsing library then has to be manually installed and managed. Consequently, updating DOMPurify itself does not update the underlying parser. This opens the door for vulnerabilities to persist, as parsing differentials in jsdom itself are no security issues. This requires users to assess the necessity for updating jsdom without any aid from the library. While bypass 1, affecting DOMPurify in its default configuration, had been independently fixed in jsdom version 20 before we were able to report it, deployment of the fix required manually updating jsdom.

We, therefore, searched for open-source projects using the vulnerable combination of DOMPurify and jsdom in version 19 to disclose our findings. This did affect projects from Mozilla and Grafana Labs, and they have resolved the issue by now.

The two libraries based on Google Caja, i.e., google-caja-sanitizer and sanitizer, are abandoned projects relying on the Caja codebase, which is itself abandoned. Consequently, reporting bugs in those libraries is infeasible, as they simply repackage the Google code. Therefore, we are currently analyzing open-source projects using a Caja-based sanitizer to see whether they are susceptible to the bypasses we found. So far, this led to a change in sanitizers in an Adobe project, but it is an ongoing effort.

5.4. Outlook

Many of the defects uncovered in the work are rooted in the overwhelming complexity of the HTML specification. While resolving them improves the state of server-side sanitization, the fundamental problem persists. This is coupled with the high rate of proposals being made toward the web platform, increasing the maintenance effort for sanitizing and parsing library authors. One recent example of this churn is the deprecation of Bleach, an HTML sanitizer for Python [48]. It relied on an unmaintained HTML parser, leading the maintainer to the conclusion that attempting to build upon an unreliable foundation is futile.

Thus, a long-term vision for input sanitization is required. Such a vision is developing on the client side, thanks to the Sanitizer API [14]. Ensuring the browser ships with a secure by default sanitizer, which guarantees to keep up with changes to the HTML and related standards, prevents a large class of XSS vulnerabilities. On the server side, such a unified solution is not feasible. Due to the heterogeneous ecosystems found on the web, a one-size-fits-all sanitizer is not possible. In addition, the update situation remains problematic, as a deployed sanitizer can get out of sync with the HTML, SVG, or MathML specification. On the client side, this is solved by automatic updates employed by all major browsers. Server-side dependency management solutions (e.g., npm) require manual intervention to install updates, with popular websites being slow to deploy new versions [49].

One helpful aspect could be to provide an HTML parsing reference implementation, usable for differential testing.[1] This would require a commitment from the browser vendors to resolve parsing divergences but would greatly simplify the validation of new parsers. Approaches such as the one presented here could then provide a large corpus of parsing edge case inputs against which new implementations can be validated. To facilitate this process, we are currently working on turning the samples with diverging behavior into tests and submitting them to the Web Platform Tests project [50] (WPT). WPT currently serves as a benchmark on how well different browsers implement various aspects of the web platform. As the major browser vendors monitor their WPT scores, this hopefully helps to shine light on these issues.

While rooting out parsing differentials reduces the likelihood of sanitizer bypasses, vulnerabilities due to logic errors will remain. As every software contains bugs, especially when dealing with a byzantine topic such as parsing HTML, a second layer of defense is required.

5.5. Mitigating Sanitizer Bypasses

Several approaches have been proposed to prevent server-side XSS vulnerabilities, including document structure integrity [51] or Noncespaces [52], both attempting to clearly differentiate user-provided content from regular markup. However, none of these proposals made it into the web platform itself.

The most realistic solution today is deploying a secure Content Security Policy (CSP) to enforce the separation of markup and code. A sufficiently strict CSP,

1. One can argue that developing a reference implementation together with updates to the specification should also improve its structure, as related information is frequently scattered across several places at the moment.

```
<script nonce="rAnd0m">g('HTML');</script>
<script>f();</script>
```

Figure 7: Two inline scripts, one with nonce and one without

which, e.g., bans inline event handlers and requires nonces or hashes to execute inline scripts, would prevent typical XSS vulnerabilities, even in the presence of a sanitizer bypass. Such a CSP realizing such a separation could look like this: `script-src 'self' https://jscdn.com 'nonce-rAnd0m';`. This policy allows loading JavaScript files from both the same origin as the site (due to the `'self'` source) as well as from jscdn.com over HTTPS. Additionally, it allows inline scripts declared with `nonce` attribute set to `r4nd0m`. Inline event handlers and scripts without a matching nonce are blocked. In Figure 7, the first script declares a nonce matching the header, and `g('HTML')` executes. The second script has no nonce attribute and is blocked due to the CSP. Such a separation requires care, however. This nonce-based approach is easily defeated by directly putting attacker-controlled input into the script's content, e.g., if an attacker can influence the value `'HTML'`.

In general, deploying secure CSPs has proven to be challenging for most websites. Difficulties stem from third-party code relying on inline scripts, forcing to forgo strict separation of markup and code by requiring directives such as `unsafe-inline`, which break the separation as shown by Steffens et al. [53]. Integrating third-party code is far from the only issue with deploying secure CSPs, as a wide range of research shows [e.g., 54–57].

5.6. Limitations & Future Work

In its current version, `MutaGen` only generates outputs containing HTML, SVG, and MathML structure. All three of these are syntactically similar. Consequently, all sanitizers process them accordingly. However, HTML has additional integration points. Both CSS (Cascading Style Sheets) as well as JavaScript can be integrated directly into HTML documents. As they are entirely different from a syntactical point of view, sanitizers must implement additional parsing modes to support this. Some of the tested sanitizers, such as AntiSamy, do this, for example, by integrating an additional parsing library for CSS. However, the interaction between these languages is also a cause for bypasses, highlighted by a recent vulnerability in rgrove/sanitize [58]. Extending `MutaGen` to generate such payloads might be an exciting opportunity for future work.

6. Related Work

We group related work into three categories: (differential) fuzzing of web technologies, differential fuzzing, cross-site scripting, and security analysis of sanitizing routines.

6.1. (Differential) Fuzzing of Web Technologies

Detecting vulnerabilities via automated test case generation is the domain of the so-called fuzz testing. When applied to the web, it is mainly used to detect memory errors inside the browser. Fuzzing JIT compilers to detect miscompilations leading to crashes and potential remote code execution vulnerabilities is a particularly active field of research [e.g., 59–61]. Similarly, the browser's HTML parser implementation can and has been tested via fuzzing, for example, by Xu et al. [62] with FREEDOM.

Semantic errors, i.e., bugs that do not manifest in crashes but unexpected or undesirable behavior, are a target less frequently considered for automated testing. This is due to fuzzing relying on so-called oracles to detect unexpected behavior. Adding an oracle to detect, e.g., buffer overflows only requires compiling the browser with modified settings. Creating an oracle detecting semantic issues is much more involved, as it requires analysis of the semantics of the application output.

One recent example where fuzzing was applied to detect semantic errors is by Kim et al. [63], who searched for universal cross-site scripting (UXSS) vulnerabilities. UXSS is universal in the sense that it does not only affect a single origin but allows the attacker to run their code in all origins.

A fuzzing technique focused on detecting divergences in behavior among different implementations for the same specification is differential fuzzing [64]. Here, inputs are generated and fed into several applications that, if correct, should behave the same. Differential fuzzing has been successfully applied to detect bugs in JavaScript JIT compilers [59], CPUs [65] and implementations of various protocols [66–68] or specifications [69]. While we consider a similar setting, applying differential testing to HTML parsing is problematic. When validating a certificate, implementations are expected to always return the same result. This is not necessarily the case for HTML parsing, as some aspects are underspecified and the negative consequences much less obvious.

6.2. Cross-Site Scripting

As the most prevalent vulnerability class on the web, XSS has undergone extensive study.

Client-side XSS is the easiest to detect, as it takes place inside the client's browser. Using a taint-tracking enabled browser, one can readily detect data flows susceptible to client-side XSS. This approach was successfully used to study the prevalence of client-side XSS [3–6, 8], improved exploit generation strategies [7] and potential defenses [70]. Similarly, Steffens et al. [71] studied the prevalence of client-side stored cross-site Scripting via dynamic taint tracking. The most related aspects to this work are those covering the generation of XSS exploit payloads [e.g., 3–7]. However, all the noted works rely on detailed insights into the application gained via taint-tracking to craft targeted exploits. Our approach, on the other hand, has no information into the inner workings of the sanitizers or the browser's HTML parser.

The complexity of HTML parsing and its impact on sanitizers has received less attention. Louw and Venkatakrishnan [72] suggested circumventing this issue by making the browser build the DOM programmatically without relying on it parsing the response in the same fashion. Simplifying the HTML specification is another seemingly attractive idea. By removing problematic tags and features, most of the issues presented in this work could be prevented. However, according to a recent study by Hantke and Stock [19], a large portion of Websites rely on HTML parsing quirks. Thus, simplifying the parsing process is not a realistic option in the near future. mXSS vulnerabilities have seen comparatively little academic attention, with only the seminal work by Heiderich et al. [25] covering it in depth. Its primary focus, however, was on browser-based mXSS vectors, while we focus on what they called "mutation based attacks" [25].

6.3. Sanitizer Analysis

A lot of work has studied the security properties of HTML sanitizers, both on the client [1, 9, 37] as well on the server-side [36, 73–75]. However, These works focus on implementation mistakes in the actual sanitizer code, i.e., by analyzing string modification chains. The bugs we consider are frequently outside the sanitizer's direct control due to the used HTML parsers returning false parsing results.

That relying on custom HTML parsing code is problematic has been highlighted by and integrated into DOMPurify [47]. However, as shown in Section 4, the chosen approach only works reliably on the client, as the sanitizer can access the browser's HTML parsing logic.

7. Conclusion

While HTML has an official specification codifying expected parsing behavior, implementing it correctly is challenging. This even affects the major browsers, which can not always agree on how a piece of markup shall be parsed. The situation is even worse for server-side HTML sanitizers despite them being an integral part of most websites' security apparatus. On the server, HTML sanitizers are fighting a losing battle, as they do not have sufficient information to accurately parse attacker-controlled input in the same way a browser does. The used parsing mode, dynamic parsing state flags, the employed browser, and its quirks are all information out of reach for the sanitizer. Lacking this information, it has to make an educated guess, frequently with devastating consequences. Parsing differentials, i.e., diverging parsing behaviors between sanitizer and browser, are one consequence of these problems and a direct security threat: Either allowing nefarious actors to bypass the sanitizer completely or to abuse the supposed protection mechanisms, making it transform benign input into harmful exploits.

In this paper, we presented `MutaGen`, a generator for mutation-prone pieces of HTML. Using `MutaGen` and our evaluation testbed, we assessed how 11 sanitizers across five programming languages deal with these kinds of inputs. Not only did we uncover functional deficiencies in each of their parsing algorithms, but we were also able to bypass all but two of them automatically. These findings highlight the sorry state of server-side HTML parsing and sanitization, a topic left unexplored for far too long.

Acknowledgments

We gratefully acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972 as well as from the European Union's Horizon 2020 research and innovation programme under project TESTABLE, grant agreement No 101019206.

References

[1] D. Klein, T. Barber, S. Bensalim, B. Stock, and M. Johns, "Hand Sanitizers in the Wild: A Large-scale Study of Custom JavaScript Sanitizer Functions," in *European Symposium on Security and Privacy*, 2022.

[2] K. Kotowicz, "Trusted types - mid 2021 report," https://research.google/pubs/pub50512, Google Research, Tech. Rep., 2021.

[3] S. Lekies, B. Stock, and M. Johns, "25 Million Flows Later: Large-scale Detection of DOM-based XSS." in *Conference on Computer and Communications Security*, 2013.

[4] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, "Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting." in *Network and Distributed Systems Security*, 2018.

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

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

[7] S. Bensalim, D. Klein, T. Barber, and M. Johns, "Talking About My Generation: Targeted DOM-based XSS Exploit Generation using Dynamic Data Flow Analysis," in *European Workshop on Systems Security*, 2021.

[8] D. Klein, M. Musch, T. Barber, M. Kopmann, and M. Johns, "Accept All Exploits: Exploring the Security Impact of Cookie Banners," in *Proc. of the Annual Computer Security Applications Conference*, 2022.

[9] D. Bates, A. Barth, and C. Jackson, "Regular Expressions Considered Harmful in Client-Side XSS Filters," in *WWW*, 2010.

[10] F. Hantke, S. Roth, R. Mrowczynski, C. Utz, and B. Stock, "Where are the red lines? towards ethical server-side scans in security and privacy research," in *IEEE Symposium on Security and Privacy*, 2024.

[11] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, E. Shin, and D. Song, "A Systematic Analysis of XSS Sanitization in Web Application Frameworks," in *ESORICS*, 2011.

[12] M. Samuel, P. Saxena, and D. Song, "Context-sensitive auto-sanitization in web templating languages using type qualifiers," in *Conference on Computer and Communications Security*, 2011.

[13] P. Saxena, D. Molnar, and B. Livshits, "SCRIPTGARD: Automatic Context-Sensitive Sanitization for Large-Scale Legacy Web Applications," in *Conference on Computer and Communications Security*, 2011.

[14] W. P. I. C. Group, "HTML Sanitizer API," https://wicg.github.io/sanitizer-api, 2022, accessed 8.12.2023.

[15] ——, "HTML Sanitizer API," https://wicg.github.io/sanitizer-api/#strings, 2022, accessed 8.12.2023.

[16] WHATWG, "HTML Standard: 1.7 Design Notes," https://html.spec.whatwg.org/#design-notes, accessed 8.12.2023.

[17] D. Megginson, "SAX," http://www.saxproject.org/, 2004, accessed: 8.12.2023.

[18] WHATWG, "HTML Standard: 13.2.10.3 Unexpected markup in tables," https://html.spec.whatwg.org/multipage/parsing.html#unexpected-markup-in-tables, accessed 8.12.2023.

[19] F. Hantke and B. Stock, "HTML Violations and Where to Find Them: A Longitudinal Analysis of Specification Violations in HTML," in *Internet Measurement Conference*, 2022.

[20] WHATWG, "HTML Standard: 13.4 Parsing HTML fragments," https://html.spec.whatwg.org/multipage/parsing.html#fragment-case, accessed 8.12.2023.

[21] T. C. Authors, "html_document_parser_fastpath.cc," https://source.chromium.org/chromium/chromium/src/+/main:third_party/blink/renderer/core/html/parser/html_document_parser_fastpath.cc, accessed 8.12.2023.

[22] W3C, "Mathematical Markup Language (MathML) Version 3.0 2nd Edition," https://www.w3.org/TR/MathML3, accessed 8.12.2023.

[23] ——, "Scalable Vector Graphics (SVG) 2," https://svgwg.org/svg2-draft, accessed 8.12.2023.

[24] WHATWG, "HTML Standard: 13.2.6.5 the rules for parsing tokens in foreign content," https://html.spec.whatwg.org/multipage/parsing.html#parsing-main-inforeign, accessed 8.12.2023.

[25] M. Heiderich, J. Schwenk, T. Frosch, J. Magazinius, and E. Z. Yang, "mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations," in *Conference on Computer and Communications Security*, 2013.

[26] D. Klein, "HTML Parsing Differentials," https://github.com/ias-tubs/HTML_parsing_differentials, 2023, accessed: 8.12.2023.

[27] M. Bentkowski, "HTML sanitization bypass in Ruby Sanitize <5.2.1," https://research.securitum.com/html-sanitization-bypass-in-ruby-sanitize-5-2-1, 2020, accessed 8.12.2023.

[28] ——, "Write-up of DOMPurify 2.0.0 bypass using mutation XSS," https://research.securitum.com/dompurify-bypass-using-mxss, 2019, accessed 8.12.2023.

[29] ——, "Mutation XSS via namespace confusion – DOMPurify <2.0.17 bypass," https://research.securitum.com/mutation-xss-via-mathml-mutation-dompurify-2-0-17-bypass, 2019, accessed 8.12.2023.

[30] E. Yalon, "Mutation Cross-Site Scripting (mXSS) Vulnerabilities Discovered in Mozilla-Bleach," https://securityboulevard.com/2020/07/mutation-cross-site-scripting-mxss-vulnerabilities-discovered-in-mozilla-bleach, 2020, accessed 8.12.2023.

[31] WHATWG, "HTML Standard: 4 The elements of HTML," https://html.spec.whatwg.org/multipage/semantics.html#semantics, accessed 8.12.2023.

[32] ——, "HTML Standard: 13.2 Parsing HTML documents," https://html.spec.whatwg.org/multipage/parsing.html, accessed 8.12.2023.

[33] ——, "HTML Standard: 4.8.5 the iframe element," https://html.spec.whatwg.org/multipage/iframe-embed-object.html#the-iframe-element, accessed 8.12.2023.

[34] ——, "HTML Standard: 3.2.5.1 the "nothing" content model," https://html.spec.whatwg.org/multipage/dom.html#the-nothing-content-model, accessed 8.12.2023.

[35] ——, "HTML Standard: 13.2.2 Parse errors," https://html.spec.whatwg.org/multipage/parsing.html#parse-errors, accessed 8.12.2023.

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

[37] M. Alkhalaf, T. Bultan, and J. L. Gallegos, "Verifying Client-Side Input Validation Functions using String Analysis," in *International Conference on Software Engineering*, 2012.

[38] M. Alkhalaf, A. Aydin, and T. Bultan, "Semantic Differential Repair for Input Validation and Sanitization," in *International Symposium on Software Testing and Analysis*, 2014.

[39] S. Joshi, N. Agrawal, R. Krishnapuram, and S. Negi, "A Bag of Paths Model for Measuring Structural Similarity in Web Documents," in *International Conference on Knowledge Discovery and Data Mining*, 2003.

[40] WHATWG, "HTML Standard: 13.2.5.2 RCDATA state," https://html.spec.whatwg.org/multipage/parsing.html#rcdata-state, accessed 8.12.2023.

[41] ——, "HTML Standard: 13.2.4.5 Parse state: Other parsing state flags," https://html.spec.whatwg.org/multipage/parsing.html#other-parsing-state-flags, accessed 8.12.2023.

[42] ——, "HTML Standard: 13.2.2 Parse errors: cdata-in-html-content," https://html.spec.whatwg.org/#parse-error-cdata-in-html-content, accessed 8.12.2023.

[43] ——, "HTML Standard: 13.2.2 Parse errors: incorrectly-closed-comment," https://html.spec.whatwg.org/multipage/parsing.html#parse-error-incorrectly-closed-comment, accessed 8.12.2023.

[44] ——, "HTML Standard: 13.2.6.4.16 The "in select" insertion mode," https://html.spec.whatwg.org/multipage/parsing.html#parsing-main-inselect, accessed 8.12.2023.

[45] T. Nidecki, "Mutation XSS in Google Search," https://www.acunetix.com/blog/web-security-zone/mutation-xss-in-google-search, 2019, accessed: 8.12.2023.

[46] W3C, "Document Structure – SVG 2," https://svgwg.org/svg2-draft/struct.html#DescriptionDefinitions, accessed 8.12.2023.

[47] M. Heiderich, C. Späth, and J. Schwenk, "DOMPurify: Client-Side Protection against XSS and Markup Injection," in *ESORICS*, 2017.

[48] W. Kahn-Greene, "bleach is deprecated; statement on project going forward (2023-01-23)," https://github.com/mozilla/bleach/issues/698, 2023, accessed 8.12.2023.

[49] N. Demir, T. Urban, K. Wittek, and N. Pohlmann, "Our (in)Secure Web: Understanding Update Behavior of Websites and Its Impact on Security," in *Passive and Active Network Measurement Conference*, 2021.

[50] web-platform-tests contributors, "The web-platform-tests project," https://github.com/web-platform-tests/wpt, accessed 8.12.2023.

[51] Y. Nadji, P. Saxena, and D. Song, "Document structure integrity: A robust basis for cross-site scripting defense." in *Network and Distributed System Security Symposium*, 2009.

[52] M. V. Gundy and H. Chen, "Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks." in *Network and Distributed System Security Symposium*, 2009.

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

[54] M. Weissbacher, T. Lauinger, and W. K. Robertson, "Why Is CSP Failing? Trends and Challenges in CSP Adoption," in *Research in Attacks, Intrusions and Defenses*, 2014.

[55] S. Calzavara, A. Rabitti, and M. Bugliesi, "Content security problems?: Evaluating the effectiveness of content security policy in the wild," in *Conference on Computer and Communications Security*, 2016.

[56] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "CSP is dead, long live CSP! On the insecurity of whitelists and the future of content security policy," in *Conference on Computer and Communications Security*, 2016.

[57] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock, "Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies," in *Network and Distributed Systems Security*, 2020.

[58] R. Grove, "Insufficient neutralization of 'style' element content may allow XSS in Sanitize," https://github.com/rgrove/sanitize/security/advisories/GHSA-f5ww-cq3m-q3g7, 2023, accessed 8.12.2023.

[59] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz, "JIT-Picking: Differential Fuzzing of JavaScript Engines," in *Conference on Computer and Communications Security*, 2022.

[60] H. Han, D. Oh, and S. K. Cha, "CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines," in *Network and Distributed System Security Symposium*, 2019.

[61] S. Groß, S. Koch, L. Bernhard, T. Holtz, and M. Johns, "Fuzzilli: Fuzzing for JavaScript JIT Compiler Vulnerabilities," in *Network and Distributed Systems Security*, 2023.

[62] W. Xu, S. Park, and T. Kim, "FREEDOM: Engineering a State-of-the-Art DOM Fuzzer," in *Conference on Computer and Communications Security*, 2020.

[63] S. Kim, Y. M. Kim, J. Hur, S. Song, G. Lee, and B. Lee, "FuzzOrigin: Detecting UXSS vulnerabilities in browsers through origin fuzzing," in *USENIX Security Symposium*, 2022.

[64] T. Petsios, A. Tang, S. J. Stolfo, A. D. Keromytis, and S. Jana, "NEZHA: Efficient Domain-Independent Differential Testing," in *IEEE Symposium on Security and Privacy*, 2017.

[65] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs," in *IEEE Symposium on Security and Privacy*, 2021.

[66] B. Jabiyev, S. Sprecher, K. Onarlioglu, and E. Kirda, "T-Reqs: HTTP Request Smuggling with Differential Fuzzing," in *Conference on*

Computer and Communications Security, 2021.
[67] G. S. Reen and C. Rossow, “DPIFuzz: A Differential Fuzzing Framework to Detect DPI Elusion Strategies for QUIC,” in *Annual Computer Security Applications Conference*, 2020.
[68] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, “Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations,” in *IEEE Symposium on Security and Privacy*, 2014.
[69] S. Wi, T. T. Nguyen, J. Kim, B. Stock, and S. Son, “DiffCSP: Finding Browser Bugs in Content Security Policy Enforcement through Differential Testing,” in *Network and Distributed System Security Symposium*, 2023.
[70] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, “Precise Client-side Protection against DOM-based Cross-Site Scripting,” in *USENIX Security Symposium*, 2014.
[71] M. Steffens, C. Rossow, M. Johns, and B. Stock, “Don’t Trust the Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild.” in *Network and Distributed System Security Symposium*, 2019.
[72] M. T. Louw and V. N. Venkatakrishnan, “Blueprint: Robust prevention of cross-site scripting attacks for existing browsers.” in *IEEE Symposium on Security and Privacy*, 2009.
[73] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, “Fast and Precise Sanitizer Analysis with BEK.” in *USENIX Security Symposium*, 2011.
[74] G. Argyros, I. Stais, A. Kiayias, and A. D. Keromytis, “Back in Black: Towards Formal, Black Box Analysis of Sanitizers and Filters,” in *IEEE Symposium on Security and Privacy*, 2016.
[75] J. Dahse and T. Holz, “Experience Report: An Empirical Study of PHP Security Mechanism Usage,” in *International Symposium on Software Testing and Analysis*, 2015.

Appendix A. Implementation

The payload generation is based on randomly selecting a sequence of transformations to consecutively build up the final payload.

Whenever a transformation or one of its parameters is selected, each option is chosen with a relative probability $\mathbb{P}$. Take the `div` and `br` tags as an example, their relative probabilities are: $\mathbb{P}(div) = 1.0$ and $\mathbb{P}(br) = 0.5$. This means `MutaGen` generates twice as many `div` tags as `br` tags.

A.1. Parameterized Transformations

Most of the transformations applied by `MutaGen` are parameterized. We now give a short overview of the different parameter types, their respective values, and how likely they are applied.

payload(): This function returns an initial payload. They are chosen from the following set: {`Img`, `Image`, `Script`} with relative probability of $\mathbb{P}(Img) = 0.6$, $\mathbb{P}(Image) = 0.2$, $\mathbb{P}(Script) = 0.2$. Each payload is serialized as follows:

- `Img`: `<img src=x onerror=f()>`
- `Image`: `<image src=x onerror=f()>`
- `Script`: `<script>f()</script>`

We decided to mainly generate XSS payloads based on `img` tags as it is the most universally applicable tag.

place(): Returns whether the transformation should change the beginning or the end of $\mathcal{P}$, returning either `Prepend` or `Append` with equal probability.

encoding(): Returns an encoding function applicable to another value. Possible values are {None, Xml} with $\mathbb{P}(None) = 0.4$ and $\mathbb{P}(Xml) = 0.1$.

quote(): This function returns an optionally encoded quote character. Possible values are chosen from: {Backtick(e), Single(e), Double(e)} where $e = encoding()$ with respective probabilities of $\mathbb{P}(Single) = 0.45$, $\mathbb{P}(Double) = 0.45$ and $\mathbb{P}(Backtick) = 0.1$.

This function is used to determine how attributes are quoted. Only single and double quotes are valid according to the specification, so they are generated more frequently.

quoted(v): Determines how an attribute’s value (provided as v) is quoted. Possible values are chosen from the set:

```
{Unquoted, Enclosed(quote()),
 Front(quote()), Back(quote()),
 Mixed(quote(), quote())}
```

`Unquoted` results in an unquoted value and `Mixed` in a value with potentially mismatching quotes, depending on the return values of its parameters. Both `Front` and `Back` result in a quote on either side of the value, and `Enclosed` properly quotes the value. Their respective probabilities are $\mathbb{P}(Unquoted) = 0.5$, $\mathbb{P}(Mixed) = 0.25$, $\mathbb{P}(Front) = 0.25$, $\mathbb{P}(Back) = 0.25$ and $\mathbb{P}(Enclosed) = 1.0$.

attr_key(): Returns a string from the set {`id`, `name`, `title`, `foo`, `data-foo`} with equal probability. We chose this selection to cover different attribute types that do not execute JavaScript on their own. We avoided generating event handlers that might directly cause JavaScript execution, as testing the completeness of block lists would offer no additional insight into the parsing behavior.

attr_form(): To represent invalid attribute values, we introduce the possibility of generating incorrectly formatted attributes. This function returns values from the set $\{Regular, Space, Slash\}$, modeling such issues. Their probabilities are $\mathbb{P}(Regular) = 0.9$, $\mathbb{P}(Space) = 0.05$ and $\mathbb{P}(Slash) = 0.05$.

attr(v): Generates a potentially quoted HTML attribute with the value v. Based on the return values of $k = attr_key()$, $f = attr_form()$ and $q = quoted()$ an attribute is serialized as follows: An attribute is serialized as follows $_k = quoted(v)$ if $f =$ `Regular`. For f equals `Space`, a whitespace character precedes the value, and if f is `Slash`, the initial space is replaced with a slash character.

tag(): Selects one of the HTML, SVG, or MathML tags listed in Table 7 with a sequence of attributes with static values. The relative probabilities for each tag are provided in column $\mathbb{P}$. These probabilities were assigned manually to group similar elements such as `mi`, `mo`, `mn`, and `ms` to uncover a wide breadth of different payloads.

bracket(): Returns either an opening or closing angle bracket with equal possibility.

bang(): Selects whether the generated XML comment should be closed according to the HTML specification (i.e., `-->`) or with a bang comment (i.e., `--!>`). Values are chosen from the set: {`No_bang`, `Bang`} with equal probability.

Table 6: Complete List of Transformations Applied to the Accumulated Payload $\mathcal{P}$

Name	$\mathbb{P}$	Parameters	Effect	Description
Payload		$pl = payload()$	$\mathcal{P} = pl$	Select an initial Payload
Open_tag	1.0	$\mathrm{t} = tag()$ $\mathrm{p} = place()$	$\mathcal{P} = \begin{cases} \texttt{<t>}\mathcal{P}, & \text{if p = Prepend} \\ \mathcal{P}\texttt{<t>}, & \text{if p = Append} \end{cases}$	Add opening tag t to $\mathcal{P}$
Self_closing_tag	1.0	$\mathrm{t} = tag(),$ $\mathrm{p} = place()$	$\mathcal{P} = \begin{cases} \texttt{<t/>}\mathcal{P}, & \text{if p = Prepend} \\ \mathcal{P}\texttt{<t/>}, & \text{if p = Append} \end{cases}$	Add self closing tag t to $\mathcal{P}$
Enclose_tag	1.0	$\mathrm{t} = tag()$	$\mathcal{P} = \texttt{<t>}\mathcal{P}\texttt{</t>}$	Enclose $\mathcal{P}$ in tag t
Enclose_tag_attr	0.75	$\mathrm{t} = tag(),$ $\mathrm{a} = attr()$	$\mathcal{P} = \texttt{<t a(}\mathcal{P}\texttt{)>}$	Enclose $\mathcal{P}$ in attribute a of tag t
Close_tag	1.0	$\mathrm{t} = tag()$ $\mathrm{p} = place()$	$\mathcal{P} = \begin{cases} \texttt{</t>}\mathcal{P}, & \text{if p = Prepend} \\ \mathcal{P}\texttt{</t>}, & \text{if p = Append} \end{cases}$	Add closing tag t to $\mathcal{P}$
Open_XML_Comment	0.125	$\mathrm{p} = place()$	$\mathcal{P} = \begin{cases} \texttt{<!--}\mathcal{P}, & \text{if p = Prepend} \\ \mathcal{P}\texttt{<!--}, & \text{if p = Append} \end{cases}$	Add opening XML comment to $\mathcal{P}$
Close_XML_Comment	0.125	$\mathrm{p} = place(),$ $b = bang()$	$\mathcal{P} = \begin{cases} \texttt{--b>}\mathcal{P}, & \text{if p = Prepend} \\ \mathcal{P}\texttt{--b>}, & \text{if p = Append} \end{cases}$	Add closing XML comment to $\mathcal{P}$
Enclose_XML_Comment	0.125	$b = bang()$	$\mathcal{P} = \texttt{<!--}\mathcal{P}\texttt{--b>}$	Enclose $\mathcal{P}$ with XML comment
Enclose_JS_Comment	0.01		$\mathcal{P} = \texttt{/*}\mathcal{P}\texttt{*/}$	Enclose $\mathcal{P}$ in JavaScript comment
Open_JS_Comment	0.005	$\mathrm{p} = place()$	$\mathcal{P} = \begin{cases} \texttt{/*}\mathcal{P}, & \text{if p = Prepend} \\ \mathcal{P}\texttt{/*}, & \text{if p = Append} \end{cases}$	Add opening JavaScript comment to $\mathcal{P}$
Close_JS_Comment	0.005	$\mathrm{p}= place()$	$\mathcal{P} = \begin{cases} \texttt{*/}\mathcal{P}, & \text{if p = Prepend} \\ \mathcal{P}\texttt{*/}, & \text{if p = Append} \end{cases}$	Add closing JavaScript comment to $\mathcal{P}$
Enclose_CDATA	0.05		$\mathcal{P} = \texttt{<![CDATA[}\mathcal{P}\texttt{]]>}$	Enclose $\mathcal{P}$ in CDATA section.
Begin_CDATA	0.05	$\mathrm{p} = place()$	$\mathcal{P} = \begin{cases} \texttt{<![CDATA[}\mathcal{P}, & \text{if p = Prepend} \\ \mathcal{P}\texttt{<![CDATA[}, & \text{if p = Append} \end{cases}$	Add begin CDATA directive to $\mathcal{P}$
End_CDATA	0.05	$\mathrm{p} = place()$	$\mathcal{P} = \begin{cases} \texttt{]]>}\mathcal{P}, & \text{if p = Prepend} \\ \mathcal{P}\texttt{]]>}, & \text{if p = Append} \end{cases}$	Add end CDATA directive to $\mathcal{P}$
Parsing_directive	0.05	$\mathrm{p} = place()$	$\mathcal{P} = \begin{cases} \texttt{<!}\mathcal{P}, & \text{if p = Prepend} \\ \mathcal{P}\texttt{<!}, & \text{if p = Append} \end{cases}$	Add incomplete parsing directive to $\mathcal{P}$
Angle_bracket	0.2	$p = place(),$ $b = bracket()$	$\mathcal{P} = \begin{cases} \mathrm{b}\mathcal{P}, & \text{if p = Prepend} \\ \mathcal{P}\mathrm{b}, & \text{if p = Append} \end{cases}$	Add angle bracket b to $\mathcal{P}$
Quote	0.25	$\mathrm{q} = quote(),$ $\mathrm{p} = place()$	$\mathcal{P} = \begin{cases} \mathrm{q}\mathcal{P}, & \text{if p = Prepend} \\ \mathcal{P}\mathrm{q}, & \text{if p = Append} \end{cases}$	Add a quote to $\mathcal{P}$
Space	1.00	$\mathrm{p} = place()$	$\mathcal{P} = \begin{cases} \textvisiblespace\mathcal{P}, & \text{if p = Prepend} \\ \mathcal{P}\textvisiblespace, & \text{if p = Append} \end{cases}$	Add a space to $\mathcal{P}$
XML_Encode	0.025		$\mathcal{P} = \texttt{xml_encode}(\mathcal{P})$	Perform XML encoding on $\mathcal{P}$
EncodeURI	0.0005		$\mathcal{P} = \texttt{encodeURI}(\mathcal{P})$	Perform URI encoding on $\mathcal{P}$
EncodeURIComponent	0.0005		$\mathcal{P} = \texttt{encodeURIComponent}(\mathcal{P})$	Perform URI Component encoding on $\mathcal{P}$
$\perp$	0.05		$\mathcal{P}$	Terminate the generation run

Table 7: Tags Generated by `MutaGen` and their Selection Criteria

Tag	$\mathbb{P}$	NS (*)	Selection Criteria
`img`		$\mathcal{H}$	Typical XSS payloads
`script`			
`image`		$\mathcal{H}$, $\mathcal{S}$, $\mathcal{M}$	In HTML treated as `img`, valid SVG or MathML element
`div`	1.0	$\mathcal{H}$	Basic HTML elements, terminate foreign content
`span`	1.0		
`object`	0.5		Basic HTML element.
`form`	1.0		`form` elements can not be nested, enforced by parsing specification
`dfn`	1.0		Both can not be nested, not enforced by parsing specification
`header`	1.0		
`p`	0.5		Optional end tag, terminates foreign content
`br`	0.5		No end tag, no content allowed, terminate foreign content
`embed`	0.5		
`input`	1.0		No end tag, no content allowed
`a`	1.0		No interactive content allowed, e.g., `iframe`, not enforced by parsing specification
`noscript`	1.0		Parsed differently depending on *scripting flag*: either HTML or JavaScript content
`table`	0.25		Opens a table, parsing specification enforces no nesting, terminates foreign content
`td`	0.25		Restrictive content, together they make up a table
`tr`	0.25		
`colgroup`	0.25		
`select`	1.0		Only `option`, `optgroup` and *script-supporting* content allowed. Special parsing rules when inside table
`option`	1.0		Restrictions on where it can occur, depending on attribute values allowed content changes
`textarea`	1.0		Only text content
`keygen`	1.0		Not supported anymore, no content, no end tag.
`xmp`	1.0		No element specification anymore, still has parsing rules, used to render markup as text without executing it
`frameset`	0.5		No element specification anymore, still has parsing rules
`listing`	1.0		No element specification anymore, still has parsing rules, used to display code
`li`	0.5		Make up a list, allowed to contain script-supporting elements, terminate foreign content
`ul`	0.5		
`pre`	1.0		Only allowed to contain phrasing content, terminate foreign content
`var`	1.0		
`dl`	0.5		Restricted content model, terminates foreign content
`dt`	0.5		Shall only occur inside `dl`, terminates foreign content
`plaintext`	1.0		Deprecated. Renders everything below as plain text. Can not be closed
`noframes`	1.0		No element specification anymore, still have parsing rules. Contain raw text content
`noembed`	1.0		
`iframe`	1.0		`iframe` element specification says no content allowed, but parsing specification says raw text content
`svg`	1.0	$\mathcal{S}$	Namespace transition from $\mathcal{H}$ to $\mathcal{S}$
`foreignObject`	1.0		Allow to embed HTML segments inside a SVG
`desc`	1.0		
`path`	1.0		
`math`	1.0	$\mathcal{M}$	Namespace transition from $\mathcal{H}$ to $\mathcal{M}$
`mtext`	0.5		Allow to embed HTML segments inside MathML
`mglyph`	0.5		
`mi`	0.25		
`mo`	0.25		
`mn`	0.25		
`ms`	0.25		
`annotation-xml`	1.0		
`style`	1.0	$\mathcal{H}$, $\mathcal{S}$	Text content when in $\mathcal{H}$, otherwise markup
`font`	1.0		Deprecated for both HTML and SVG
`title`	1.0	$\mathcal{H}$	Text content, Singleton: not enforced by parsing specification
		$\mathcal{S}$	Can contain markup

*: $\mathcal{H}$: HTML namespace, $\mathcal{S}$: SVG namespace, $\mathcal{M}$: MathML namespace

Appendix B.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

The paper conducts an analysis of server-side HTML sanitization and parsing libraries and their vulnerabilities. They evaluated 11 such libraries using their HTML fragment generator MutaGen and uncovered security issues in nine of them. The authors then categorize the root causes of these vulnerabilities into five main parsing issues and two serialization problems.

B.2. Scientific Contributions

- Identifies an Impactful Vulnerability
- Creates a New Tool to Enable Future Science

B.3. Reasons for Acceptance

1) Identifies an Impactful Vulnerability: This study offers an examination of issues arising in sanitization libraries as a result of incorrect parsing of HTML snippets. Their findings show the existence of HTML parsing and sanitization flaws that can lead to significant security vulnerabilities, as evidenced by the presence of CVEs.
2) Creates a New Tool to Enable Future Science: MutaGen or the design idea behind the tool might be interesting for future research, e.g. altering the tool to focus on stylesheet injections instead of XSS.

B.4. Noteworthy Concerns

1) The paper does not adequately explain the criteria for selecting the analyzed sanitizers. The current selection could be biased, and the results may not represent server-side sanitizers that are actually used in the wild.
2) Some reviewers raised concerns that the approach does not consider CSS injections.