Talking About My Generation: Targeted DOM-based XSS Exploit Generation using Dynamic Data Flow Analysis

Souphiane Bensalim
SAP Security Research
souphianebensalim@gmail.com

David Klein
Technische Universität Braunschweig
david.klein@tu-braunschweig.de

Thomas Barber
SAP Security Research
thomas.barber@sap.com

Martin Johns
Technische Universität Braunschweig
m.johns@tu-braunschweig.de

ABSTRACT

Since the invention of JavaScript 25 years ago, website functionality has been continuously shifting from the server-side to the client-side. Web browsers have evolved into an application platform, and HTML5 emerged as a first-class environment for building rich cross-platform applications. This additional functionality on the client-side comes with the added risk of new security issues with increasingly severe consequences. In this work, we investigate the prevalence of DOM-based Cross-Site Scripting (DOM-based XSS) in the top 100,000 most popular websites using a novel targeted exploit generation technique based on dynamic data-flow tracking. In total, this work finds 15,710 potentially insecure data-flows where information from the URL is injected into the HTML of the Web page. Using large-scale exploit generation and validation services, 7199 of these flows lead to JavaScript execution, across 711 different domains. This represents a successful exploit rate of 45.82%, improving on previous methods by factors of 1.8 and 1.9 respectively.

CCS CONCEPTS

• **Security and privacy → Browser security**.

KEYWORDS

Web Security, DOM-based XSS, Exploit Generation, Taint Tracking

ACM Reference Format:
Souphiane Bensalim, David Klein, Thomas Barber, and Martin Johns. 2021. Talking About My Generation: Targeted DOM-based XSS Exploit Generation using Dynamic Data Flow Analysis. In *14th European Workshop on Systems Security (EuroSec'21), April 26, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3447852.3458718

1 INTRODUCTION

The advent of Web 2.0 marked a new era of dynamic Web applications, leading to a shift from a static to an interactive Web. Client-side code gained more importance and grew in complexity, with Web browsers becoming an application platform which executes JavaScript code dynamically while documents are being displayed. This change in the architecture of Web applications has brought with it severe security implications on the client-side. As a consequence, the number of security vulnerabilities in Web applications has increased significantly in parallel to the evolution of Web technologies [1]. An example of the security issues that arose with the new capabilities of Web 2.0 is related to the `<iframe>` tag used to embed another document within the current document. This tag has been shown to have several information leakage issues across nested browsing contexts, which have lead to the introduction of several security policies [13]. The best known of these policies is the Same-Origin Policy (SOP) that defines resource access controls in Web applications: *A resource can access another resource only if the two share the same origin* [20].

To bypass these constraints, malicious attackers found new ways to execute malicious code in the origin of vulnerable webpages, e.g., via so-called Cross-Site Scripting (XSS). This type of code injection attack is used by an attacker to interact with the vulnerable Web application in the name of the victim, with the main aim of stealing sensitive session data and cookies. Until 2005, there were only two kinds of Cross-Site Scripting: stored-XSS and reflected-XSS. By 2005, and due to the shift of Web applications to the client-side, Klein [7] introduced DOM-based XSS as the third kind of Cross-Site Scripting, which occurs purely inside client-side code.

Methods to detect and validate DOM-based XSS were first introduced by Lekies et al. [8] and later improved by Melicher et al. [9]. Their techniques for URL exploit injection, are based on heuristic assumptions about how URLs are processed by client-side JavaScript code. In this paper, we aim to improve on these techniques by introducing a targeted exploit generation technique to ensure payloads are injected into a URL in the position where they have the highest chance of triggering JavaScript execution. To achieve this goal, we collected potentially vulnerable data-flows from the 100,000 top ranked domains according to the Tranco [12] list. Next, we generated exploits using our novel exploit generation technique and validated these exploits. To assure a fair comparison, we also implemented the approaches from the past works and generated exploits for the same collection of domains.

This paper is organized as follows: First DOM-based XSS is briefly presented in Section 2 followed by an overview of the approach taken in Section 3. Existing techniques for exploit generation are presented in Section 4, succeeded by the introduction of a novel

EuroSec'21, April 26, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8337-0/21/04…$15.00
https://doi.org/10.1145/3447852.3458718

targeted exploit method in Section 5. Results are presented and discussed in Section 6, followed by related work in Section 8 and conclusions in Section 9.

2 BACKGROUND

Cross-Site Scripting is a type of security vulnerability in which malicious scripts are injected into a Web application in order to transfer sensitive data to a malicious third party [19]. Programming errors in trusted websites are exploited to perform a script execution within the Web browser of the victim, allowing the attacker to perform various nefarious actions, including stealing cookies and sensitive information, keylogging, session hijacking and so on.
In this section, we provide a brief summary of DOM-based XSS, followed by a classification of the different contexts which can lead to the execution of malicious scripts in client-side code.

2.1 DOM-based XSS

DOM-based Cross-Site Scripting is a special case of XSS which appeared as Web applications have transitioned from traditional static webpages to feature-rich client-side JavaScript rendering in the Web browser. In DOM-based XSS, the script execution occurs only while a webpage is being loaded or after it is loaded, which means that the malicious script is not inserted into the webpage on the server side but on the client-side. Here, the Web browser does not differentiate between a safe script issued from the developer and a malicious script injected by an attacker. In order to successfully exploit this type of vulnerability, an attacker builds a crafted URL containing the payload and delivers it to a victim e.g., via e-mail.
DOM-based XSS is caused by insecure data-flows present in client-side code in which data from user-controlled *source* functions flows into functions which allow script execution in the DOM (known as *sink* functions). Examples of relevant sources for this work are `location` and `document.URL` properties. Corresponding examples for sink functions and properties that allow the programmers to dynamically change the DOM (and therefore allow an attacker to inject malicious scripts) are: `document.write`, `innerHTML`, `eval` and attribute event handlers. In the following, we describe the various sink function categories in more detail.

2.2 Script Execution Contexts

Determining the script execution context is crucial for effective exploit generation, as each context has different exploitation criteria. Specifically, we divide the execution contexts into five categories as follows:

(1) **HTML Context:** It is possible for Web applications to inject HTML code directly into the document of the page. In case the injected code originates from a user-controlled source, DOM-based XSS is possible. Within the HTML context we differentiate between the injection into the HTML content and the injection into the HTML attributes as follows:
  - HTML content: JavaScript allows the dynamic creation and modification of DOM Elements on a webpage. JavaScript can be embedded directly in some methods, such as `document.write` and `document.writeln`, by simply adding a `<script>` tag. Other sinks, such as `innerHTML` and `insertAdjacentHTML` insert, but do not execute the script, but can still be exploited using Event Handlers, as described in Section 4.2.
  - HTML attributes: An HTML attribute is set and modified via JavaScript in two manners: Either by using the method `elem.setAttribute(name, value)` or by using string concatenation. This makes a flow from a user-controllable source to an attribute value possible as follows:
    - If the attribute is an HTML5 Event Handler, `href` or `src` attribute: The code can be injected directly into the attribute and will be executed when the event occurs.
    - Otherwise: The context should be first closed in order to inject either an Event Handler inside the same tag or reach an HTML Content context and inject a script.

  The attribute context is divided into three sub-contexts: Single Quoted attributes, Double Quoted attributes and Unquoted attributes.

(2) **URL Context:** In the URL-Context, a user-controlled input is injected into a URL attribute of certain DOM elements such as the `href` attribute of an anchor tag, the `src` attribute of an image, iframe or embed tag, and the `data` attribute of the object tag. Even if HTML encoding prevents breaking out of the HTML context, there is still a potential threat. The attacker can, for example, make use of the `javascript:`, `data:` or `vbscript:` scheme to run the malicious script if they are able to control the whole attribute.

(3) **JavaScript Context:** Web applications which turn user provided input into executable code, e.g., via `eval()`, `Function()`, `setTimeout()`, or `setInterval()` are also at risk of DOM-based XSS. These functions take the code in string format and execute it as JavaScript. Template literals are another type of JavaScript context introduced in ECMAScript6, consisting of string literals encapsulated in backticks. Template literals allow embedded JavaScript expressions using the `${...}` syntax to be evaluated and inserted in the DOM.

(4) **JSON Context:** In the JSON Context, a user-controllable input is reflected as a JavaScript Object Notation (JSON) value. JSON is a standard format used to represent structured data as attribute-value pairs and array data types. It is commonly used to serialize and transmit data within and between Web applications. This context can be exploited by breaking out from the JSON context into the JavaScript context and then injecting malicious code.

(5) **CSS Context:** Cascading Style Sheets (CSS) provide a mechanism to add style to Web applications. In the CSS Context, a user-controllable input is reflected into the value of a style attribute or tag. This context is only relevant for XSS exploits in legacy browsers, e.g., Internet Explorer, where JavaScript is allowed to be part of the CSS value.

The contexts presented above are classified into three sink categories: *HTML element sink* (consisting of Items 1 and 2), *execution sink* (Items 3 and 4) and *CSS sink* (Item 5). As described above, CSS sinks are considered out of scope for this work.

3 OVERALL APPROACH

In this section, we describe an automated system for the detection of potentially insecure data-flows in Web applications and how

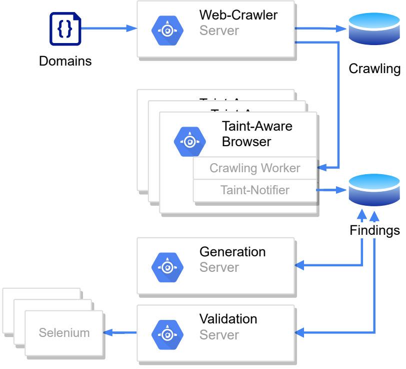

Figure 1: Overview of the crawling, exploit generation and validation architecture

they can be used for the generation and validation of DOM-based XSS exploits. The approach is summarized in Figure 1 and consists of three phases:

(1) **Detection**: The crawler visits specific web pages to collect potentially insecure data-flows.
(2) **Generation**: Exploit URLs are generated from the appropriate data-flows which could lead to a DOM-based XSS exploit.
(3) **Validation**: Generated URLs are tested to see whether they lead to arbitrary JavaScript execution.

More details on the detection and validation phases is given in Section 3.1 and Section 3.2 respectively. As exploit generation forms the focus of this work, complete sections are dedicated to the descriptions of existing techniques (Section 4) and a novel targeted injection method (Section 5).

3.1 Detection of Insecure Flows

In order to detect insecure DOM-based XSS data flows from sources to sinks, we employed a dynamic taint-tracking approach by directly instrumenting the Web browser. To build this browser, we used the open-source Firefox browser and enhanced its JavaScript engine *SpiderMonkey* as well as the DOM implementation with functions that allow the taint-tracking of all the strings present in a web page. Every strings emanating from a source function is *tainted* by attaching rich character-level metadata describing the sink function and its location in the code. Any string transformation operations, such as `substring`, `concat` and `replace` are also enhanced so that the taint metadata is correctly propagated. In addition, a record of each string operation and its code location is attached to the metadata, allowing full reconstruction of the taint operations performed. If a string with one or more tainted characters is detected entering a sink function, a JavaScript event is triggered in the browser, which is forwarded to an external database by a dedicated web extension. For our study we considered the same set of sources and sinks as the previous DOM-based XSS studies [8, 9].

The large-scale collection of taint flows is automated by embedding our taint-aware browser in a Selenium-driven Docker container. A central Web crawling service takes an input of seed domains which are distributed to a scalable number of workers. Each worker will load its designated URL before extracting all links present on the page, which are sent back to the server and the worker is ready to receive the next URL. Each worker waits for 10 seconds after a successful page load in order to maximize the number of taint flows collected. If an error occurs during loading, or the page is not loaded after 30 seconds, an error or timeout status is returned to the server respectively. The crawling server can be configured to visit a fixed number of links per page, up to a certain depth from the starting domain.

3.2 Exploit Validation

In order to explore the effectiveness of the exploits generated in Section 4.2, we check whether they lead to JavaScript execution in the validation stage. Validation is performed by a headless Firefox browser, driven by Selenium and running inside a Docker container. Each validation instance retrieves an unvalidated exploit URL from the database, with a dedicated listener installed to detect execution of the *payload* function. If the function is called, then the exploit is considered successful and the result written back to storage. Following a similar approach to Lekies et al. [8], we use a modified Firefox browser which does not automatically encode the `search` and `fragment` portions of the URL, in order to mimic the behavior of legacy browsers such as Internet Explorer. In addition, each exploit is opened in a fresh browser instance to minimize effects due to, e.g., cookie storage.

4 ESTABLISHED EXPLOIT GENERATION

Existing methods for DOM-based XSS exploit generation are presented in this section, including an overview of exploit generation (Section 4.1), followed by URL injection techniques (Section 4.2).

4.1 Context-sensitive Exploit Generation

In order to exploit a DOM-based XSS vulnerability, an attacker has to build a URL containing a crafted malicious exploit. In general, an XSS exploit consists of three parts as follows:

$$exploit := breakOut + payload + breakIn$$

The first part is the *breakOut* sequence whose purpose is to "break out" of a non-executable context to a context where JavaScript can be executed. The second part is the *payload* consisting of the script code that has to be executed. The third part is the *breakIn* sequence which serves to escape any subsequent code sequences in order to prevent them from causing execution errors.

The break out sequence is generated following a similar method to Lekies et al. [8], using a parser to identify the context (as described in Section 2.2) where the tainted part has been injected and therefore generate context-dependent closing characters. For HTML element sinks, some tags such as `<textarea>` prevent a script execution inside them. As the string flowing into a sink function will typically only contain a fraction of the entire HTML code, we cannot determine whether such a tag has previously been opened. In order to counteract this effect, the *breakOut* sequence is appended with a list of closing tags as follows:

```
</iframe></style></script></object></embed></textarea>
```

This is only possible due to the fault tolerance of the HTML parser, which allows the presence of closing tags even without a preceding opening tag.

More care needs to be taken with the JavaScript parser, which blocks code execution if a syntax error is detected. Therefore, the generated exploit should replace the tainted portion of the string without causing any syntax errors (for example by forgetting to close a bracket or closing an unopened context). To minimize these errors, we enhanced the parsing step with an error-correcting syntax check to ensure execution will not be canceled.

After breaking out of the sink context to a context where a script can be executed, the next step is to construct the target *payload*. For this study, the payload is simply a custom function call, whose execution is monitored as part of the validation stage. While an execution sink represents a script context and therefore a direct call to the function is possible, the HTML element sink will interpret this call as text and will not execute it directly. In this case it is necessary to first enter a script context as follows:

`<script>reportingFunction()</script>`

This technique succeeds for methods such as `document.write` and `document.writeln`. However, HTML5 does not allow direct script execution using the following properties: `innerHTML`, `outerHTML` and `insertAdjacentHTML`. For these cases script execution is still possible via an event handler as follows:

`<img src=x onerror="reportingFunction()">`

In this case, the image source `x` is deliberately unavailable, triggering execution of the `onerror` event handler.

Finally, the *breakIn* sequence serves to close the execution context of the payload and then comment out any characters proceeding it. This ensures that the parser will not abort the script execution due to a syntax error.

Listing 1: location.hash flows to the JavaScript context

```
let hash = location.hash; // source
if (hash.length > 1) {
    let username = unescape(hash.substr(1));
    let editUsername = "<input type="text" value=\""
        + username + "\" />";
    document.getElementById("msgboard").innerHTML =
        editUsername; // sink
}
```

Listing 2: context-specific generated exploit

```
"><img src=x onerror="reportingFunction()"><!--
```

Listing 1 presents an example where data flows from `location.hash` into `innerHTML`. The context is the HTML element sink, or more precisely the HTML attribute context. A valid exploit generated by the method described above is presented in Listing 2, where `">` is the break out sequence that closes the attribute context, followed by an appropriate `innerHTML` payload, with `<!--` providing the break in sequence.

4.2 URL Injection Techniques

Once an exploit was created, it must be inserted into the URL in a position which is most likely to result in the payload's execution. In this Section, we present two previous techniques which address this issue, and present a novel method for targeted exploit injection. Lekies et al. [8] proposed a method in 2013 (termed *Method A* in the following) that consists of appending the URL with a *fragment* (or *hash*) containing the generated exploit. As the fragment portion of the URL is not sent as part of the HTTP request, this method has the advantage of assuring that the server will not know that a URL with a malicious script has been requested. In 2018, Melicher et al. [9] presented a complementary approach (termed *method B*) that treats query parameters as a special case, making it more specific than method A. In their work, methods A and B were subsequently combined to achieve a larger number of exploit candidates. Method B consists of moving the query string in question to the fragment and change its value to an exploit.

To highlight how methods A and B work, we provide an example of a URL where a vulnerable flow has been identified coming from the `location.search` source:

`http://example.com/1?payload=`**`abcd`**`&sp=x` (1)

With method A, the exact location of the tainted string **abcd** within the URL is not considered and a context-sensitive generated exploit is independently appended as follows:

Method A:
`http://example.com/1?payload=`**`abcd`**`&sp=x#`**`EXPLOIT`**

In comparison, method B determines whether the tainted string appears within a query parameter, as is the case in the example above. Method B will then extract the corresponding key-value pair from the query and replace the complete value with the generated exploit, for example as follows:

Method B: `http://example.com/1?sp=x#&payload=`**`EXPLOIT`**

Based on the results of [9], URL parameters are commonly extracted in client-side JavaScript code by searching for characters like ?, & and =. For this reason, a & character is added to the extracted key-value pair at the beginning of the exploit.

5 TARGETED EXPLOIT GENERATION

Both of the existing methods, however, are based on heuristic assumptions about how the URL is processed by client-side code, and thus suffer from a level of imprecision. To address this issue a novel, targeted exploit injection technique (termed *Method C*) is presented in this section. Method C uses the information about the tainted flow to resolve the context where it appears within the URL, and defines the corresponding range of characters to be replaced.

With method C, the tainted part (i.e., **abcd**) of URL 1 is interpreted as a query parameter value, and will be completely replaced by the exploit as follows:

Method C: `http://example.com/1?payload=`**`EXPLOIT`**`&sp=x`

In general the tainted characters may span multiple components of the URL, so it is important to know where exactly the tainted string is present in the source (i.e., the URL) and also in the sink. To illustrate this, consider the following URL, where the source is `location.href` and the tainted string is `1?payload=abcd`:

`http://example.com/`**`1?payload=abcd`**`&sp=x` (2)

Replacing the entire tainted string with the generated exploit will cause the URL to redirect to a different location (which may no longer contain the desired data-flow). To solve this, we define six indices relevant for the exploit generation as follows:

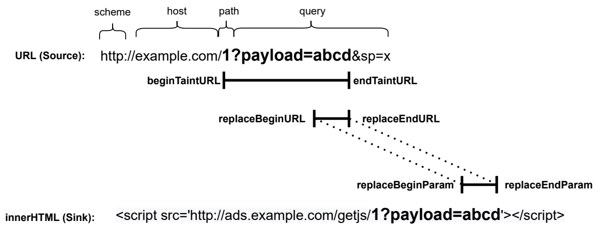

Figure 2: An example of targeted exploit injection

- **beginTaintURL** and **endTaintURL**: Represent the start and the end indices of the tainted string in the URL, and is computed by searching for the tainted string within the URL.
- **replaceBeginURL** and **replaceEndURL**: Represent the start and the end indices of the part of the tainted string that can be replaced without causing the application to perform in an unexpected way.
- **replaceBeginParam** and **replaceEndParam**: These are the corresponding indices for *replaceBeginURL* and *replaceEndURL* within the sink context and are used in the exploit generation to analyze the preceding code and generate a correct *breakOut* (Section 4).

In a final step, the generated exploit is inserted into the URL, replacing the characters between the **replaceBeginURL** and **replaceEndURL** indices.

An example for the URL in 2 is shown in Figure 2. The tainted string `1?payload=abcd` is first identified in the URL and the corresponding computed values for **beginTaintURL** and **endTaintURL**. The string `abcd` is identified as replaceable, with indices of **replaceBeginURL** and **replaceEndURL**, which correspond to **replaceBeginParam** and **replaceEndParam** of the `innerHTML` sink string. This position is used to generate a context-specific exploit, replacing **replaceBeginURL** and **replaceEndURL** in the URL.

While the example shows replacement of a query parameter, it will also successfully replace characters in other regions of the URL, for example in the fragment.

The main advantage of method C is that the exploit is injected into the URL in a position where it is most likely to result in successful code execution. For example, consider the case where the URL query contains a JSON encoded data structure. URLs generated by methods A and B will simply replace the tainted portion of the string, resulting in invalid JSON and causing the program to error before the payload can reach the sink. In comparison, method C will replace the appropriate parameter of the data structure, producing valid JSON and successful code execution.

In addition to targeted URL insertion, method C offers several advantages over existing A and B. For example, if the `decodeURI` or `decodeURIComponent` is detected as part of the taint flow, the corresponding encoded string will be searched for and replaced in the exploit URL.

6 RESULTS AND DISCUSSION

Using the techniques described above, large-scale exploit detection, generation and validation was performed on real-world websites, with two goals: to re-measure the performance of existing exploit generation techniques on today's websites, and to assess the effectiveness of the new method described in Section 5. The study was carried out in two phases: first, a list of the 100,000 top ranked domains by Tranco [12] was crawled[1], and the corresponding taint flows collected. The second phase consisted of selecting the relevant flows from the first phase, generating the exploits using the three different methodologies and validating these exploits.

Table 1: Crawling results comparison with previous studies

	This work C	DOMsday B [9]	25m Flows A [8]
Date	26/10/2020	01/08/2017	04/11/2013
Seed domains	100,000	10,000	5000
Sub-pages up to	10	5	all depth 1
Web pages	390,092	44,722	504,275
Pages/Domain	3.90	4.47	100.86
Frames	1,111,821	319,481	4,358,031
Taint Flows	20,912,107	4,140,873	24,474,873
Flows/Page	53.61	92.59	48.53
Flows/Frame	18.81	12.96	5.62

Each top-level domain was crawled, and up to 10 links hosted on the same domain as the starting page were selected for further visits. A total of 390,092 pages were visited from 100,000 domains over 4 days in October 2020. Table 1 presents a comparison between the crawling results done in this work versus the crawling results from the studies of Lekies et al. [8] and Melicher et al. [9].

Following the methodology of [8, 9], we first filter the tainted flows to select only those with URL-based sources and HTML or JavaScript sinks (*F1*). Flows containing `escape`, `encodeURI` or `encodeURIComponent` operations without a subsequent `unescape`, `decodeURI` or `decodeURIComponent` are also excluded (*F2*), as these functions encode characters which are necessary for exploit execution (e.g., < is encoded as `%3C`). Duplicate flows are removed as well applying the prescription of [8] based on the domain, code location and breakout sequence (*F3*). The number of flows after each filter is as follows:

$$20{,}912{,}107 \xrightarrow{F1} 338{,}906 \xrightarrow{F2} 227{,}510 \xrightarrow{F3} 15{,}710$$

Exploits were generated using each method for the remaining 15,710 flows and their effectiveness validated. An overview of exploit validation results for each method is shown in Figure 3.

Overall, 7199 out of 15,710 (45.82%) of flows could be successfully validated using method C, across 711 unique domains. This success rate represents an improvement of 1.9 over method A (24.43%) and 1.8 over method B (25.72%). In addition, 846 (11.15%) of flows could only be validated using method C, and were not successful with either of the previous methods. As method C replaces characters in the URL directly with an exploit, it is more likely to reach the corresponding sink function without causing decoding or parsing errors. Overall, 7588 (48.30%) unique flows out of 15,710 could be successfully validated when combining all three methods.

[1] Available at: https://tranco-list.eu/list/K9ZW, list downloaded on 7th October 2020.

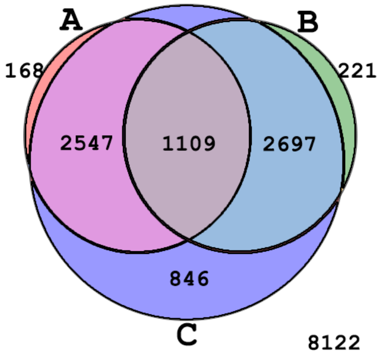

Figure 3: Venn diagram of the validated flows using methods A (25m flows), B (DOMsday) and C (this work)

A small fraction of flows could not be validated by method C, but were successful with the other methods: 168 flows could only be validated with method A (2.21% of all validated flows), and 221 flows could only be validated with method B (2.91% of all validated flows). After manually analyzing these cases, we found that the URL generated by method C in some situations breaks the application logic, e.g., when a parameter value was used to redirect the visitor to a specific page.

7 LIMITATIONS

A general issue when validating exploit URLs is the issue of concept drift. When loading a dynamic website several times, its content frequently changes between visits, e.g., due to different advertisements being served. Some vulnerable flows are therefore invisible on subsequent attempts to visit the website.

Modern Web browsers implement URL-Encoding as required by RFC 1738 [3] to encode some reserved characters that might be used in an unsafe way. While this mechanism allows parsing the URL components correctly, it also defends against some XSS attacks as the exploit injected into the URL is subsequently encoded. In this work, we disabled the URL encoding mechanism firstly for comparison reasons with the two previous works and secondly because websites shouldn't rely on an external security mechanism. Internet Explorer for example doesn't implement any URL-Encoding and at the time of writing still holds a market share of 1,89% for desktop browsers worldwide [14].

In addition, we consider only first level flows, that is flows that directly go into a sink allowing script execution. Second level flows, e.g., flows where a tainted value is stored in a cookie or local storage and later inserted into a sink (potentially on a different part of the website) are out of scope for this work. We refer to the work by Steffens et al. [15] for an investigation of such second level flows.

One potential drawback of method C is that changes to the query portion of the URL are sent to the server (changes to the fragment are not), which could be detected and blocked by server-side defense mechanisms, such as Web Application Firewalls (WAFs). The fragment can also be transmitted to the server for methods A and B, however, as it can be packaged into an XMLHttpRequest (XHR). This is especially relevant for single-page applications, which make extensive use of XHR.

8 RELATED WORK

DOM-based XSS vulnerabilities have been a topic of active research over the last decade. While Amit Klein initially coined the term in 2005 [7], dynamic taint tracking to detect DOM-based XSS vulnerabilities has been used since 2012 when Di Paola introduced DOMinator [4], a Firefox Extension able to detect such vulnerabilities. Lekies et al. [8] were the first to investigate the prevalence of this vulnerability class on a large scale. Melicher et al. [9] did improve on their methodology in 2018 and confirmed that DOM-based XSS vulnerabilities remain a highly relevant issue.

DOM-based XSS studies have become increasingly elaborate over time. Building on the work of Lekies et al. [8], Stock et al. [16] studied their history as well as the code patterns resulting in vulnerabilities [18] while Steffens et al. [15] investigated persistent DOM-based XSS vulnerabilities. To overcome the performance impact of taint tracking, Melicher et al. [10] suggested a method where tainting is only applied to websites suspected to contain vulnerabilities, based on a Machine Learning prefilter.

In addition to studying the prevalence of DOM-based XSS vulnerabilities, several attempts have been made to protect websites. Chrome as well as Internet Explorer used to include XSS Filter mechanisms, which inspected the data and blocked suspicious requests. Due to dynamic content generation being the foundation of the Web 2.0, the high number of false positives lead ultimately to their removal [2, 17]. Attempts to automatically retrofit sanitization functions before calling sinks, as attempted by Musch et al. in ScriptProtect [11] breaks a significant portion of the websites in question.

Browser vendors are currently working on two promising approaches to counteract DOM-based XSS vulnerabilities. Trusted Types [5] proposes integration of a policy mechanism into the type system. If the developer wants to turn text into a DOM element, the text has to conform to the given policy, e.g., not contain any markup. Another proposal is to include a sanitizer API directly in the browser. This would allow developers to protect a Web application without having to rely on the error prone process of writing their own sanitization functionality or having to introduce additional external libraries such as DOMPurify [6].

9 CONCLUSION

We investigated the top 100,000 websites and analyzed them for DOM-based XSS vulnerabilities using the presented methodology to generate and validate exploits. In total, 7199 exploits have been successfully validated out of the 15,710 relevant for an exploit generation, representing a success rate of 45.82%. This represents an improvement over the two previous exploit generation techniques of [8, 9] by factors of 1.9 and 1.8 respectively. In addition, the new technique presented here successfully generated exploits for 846 (11.15%) URLs, which were not successful with either of the previous methods.

Despite rising of awareness about the dangers of DOM-based XSS vulnerabilities over the last decade many of the top ranked websites still suffer from such vulnerabilities.

ACKNOWLEDGMENTS

We acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and from the state of Lower Saxony under the project Mobilise.

REFERENCES

[1] 2020. Edgescan: Vulnerability Stats Report. *Network Security* 2020, 3 (2020), 4. https://doi.org/10.1016/S1353-4858(20)30027-1

[2] Daniel Bates, Adam Barth, and Collin Jackson. 2010. Regular Expressions Considered Harmful in Client-Side XSS Filters. In *Proceedings of the 19th International Conference on World Wide Web* (Raleigh, North Carolina, USA) *(WWW '10)*. Association for Computing Machinery, New York, NY, USA, 91–100.

[3] Tim Berners-Lee, Larry Masinter, and Mark McCahill. 1994. RFC1738: Uniform Resource Locators (URL).

[4] Stefano Di Paola. 2012. DominatorPro: Securing Next Generation of Web Applications. http://web.archive.org/web/20120712202421/https://dominator.mindedsecurity.com//.

[5] Web Incubator Community Group. 2017. Explainer: Trusted Types for DOM Manipulation. https://github.com/WICG/trusted-types.

[6] 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, Cham, 116–134.

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

[8] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: Large-scale detection of DOM-based XSS. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 1193–1204.

[9] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out domsday: Towards detecting and preventing dom cross-site scripting. In *2018 Network and Distributed System Security Symposium (NDSS)*.

[10] William Melicher, Clement Fung, Lujo Bauer, and Limin Jia. 2021. Towards a Lightweight, Hybrid Approach for Detecting DOM XSS Vulnerabilities with Machine Learning. In *Proceedings of The Web Conference*. To appear.

[11] Marius Musch, Marius Steffens, Sebastian Roth, Ben Stock, and Martin Johns. 2019. ScriptProtect: Mitigating Unsafe Third-Party JavaScript Practices. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security* (Auckland, New Zealand) *(Asia CCS '19)*. Association for Computing Machinery, New York, NY, USA, 391–402.

[12] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation.. In *Proc. of Network and Distributed System Security Symposium (NDSS)*.

[13] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. 2017. Same-origin policy: Evaluation in modern browsers. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 713–727.

[14] statcounter. 2021. Desktop Browser Market Share Worldwide | StatCounter Global Stats. https://gs.statcounter.com/browser-market-share/desktop/worldwide. (Accessed on 03/28/2021).

[15] 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 *Proc. of Network and Distributed System Security Symposium (NDSS)*.

[16] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. 2017. How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security.. In *Proc. of USENIX Security Symposium*. 971–987.

[17] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. 2014. Precise Client-side Protection against DOM-based Cross-Site Scripting.. In *Proc. of USENIX Security Symposium*. 655–670.

[18] Ben Stock, Stephan Pfistner, Bernd Kaiser, Sebastian Lekies, and Martin Johns. 2015. From Facepalm to Brain Bender: Exploring Client-Side Cross-Site Scripting.. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 1419–1430.

[19] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2007. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis.. In *NDSS*, Vol. 2007. 12.

[20] Michal Zalewski. 2012. *The tangled Web: A guide to securing modern web applications.* No Starch Press.